Chapter Objectives

- To learn how to use a tree to represent a hierarchical organization of information
- To learn how to use recursion to process trees
- To understand the different ways of traversing a tree
- To understand the differences between binary trees, binary search trees, and heaps
- To learn how to implement binary trees, binary search trees, and heaps using linked data structures and arrays
Chapter Objectives (cont.)

- To learn how to use a binary search tree to store information so that it can be retrieved in an efficient manner
- To learn how to use a Huffman tree to encode characters using fewer bits than ASCII or Unicode, resulting in smaller files and reduced storage requirements
Trees - Introduction

- All previous data organizations we've studied are linear—each element can have only one predecessor and successor
- Accessing all elements in a linear sequence is $O(n)$
- Trees are nonlinear and hierarchical
- Tree nodes can have multiple successors (but only one predecessor)
Trees - Introduction (cont.)

- Trees can represent hierarchical organizations of information:
  - class hierarchy
  - disk directory and subdirectories
  - family tree
Trees are recursive data structures because they can be defined recursively.

Many methods to process trees are written recursively.

This chapter focuses on the binary tree.

In a binary tree each element has two successors.

Binary trees can be represented by arrays and by linked data structures.

Searching a binary search tree, generally is more efficient than searching an ordered list—$O(\log n)$ versus $O(n)$. 
Tree Terminology and Applications

Section 8.1
A tree consists of a collection of elements or nodes, with each node linked to its successors.
A tree consists of a collection of elements or nodes, with each node linked to its successors.

The node at the top of a tree is called its root.
A tree consists of a collection of elements or nodes, with each node linked to its successors.
A tree consists of a collection of elements or nodes, with each node linked to its successors.

The successors of a node are called its children.
Tree Terminology (cont.)

A tree consists of a collection of elements or nodes, with each node linked to its successors.

The predecessor of a node is called its parent.
A tree consists of a collection of elements or nodes, with each node linked to its successors.

Each node in a tree has exactly one parent except for the root node, which has no parent.
Tree Terminology (cont.)

A tree consists of a collection of elements or nodes, with each node linked to its successors.

Nodes that have the same parent are siblings.
Tree Terminology (cont.)

A tree consists of a collection of elements or nodes, with each node linked to its successors.

A node that has no children is called a leaf node.
A tree consists of a collection of elements or nodes, with each node linked to its successors.

A node that has no children is called a *leaf node*.

Leaf nodes also are known as *external nodes*, and nonleaf nodes are known as *internal nodes*.
A tree consists of a collection of elements or nodes, with each node linked to its successors.
A tree consists of a collection of elements or nodes, with each node linked to its successors.

Dog is the parent of cat in this tree.

A generalization of the parent-child relationship is the ancestor-descendant relationship.
A tree consists of a collection of elements or nodes, with each node linked to its successors.

- Cat is the parent of canine in this tree.
- A generalization of the parent-child relationship is the ancestor-descendant relationship.
A tree consists of a collection of elements or nodes, with each node linked to its successors.

- canine is a descendant of cat in this tree.
- A generalization of the parent-child relationship is the ancestor-descendant relationship.
Tree Terminology (cont.)

A tree consists of a collection of elements or nodes, with each node linked to its successors.

- Dog is an ancestor of canine in this tree.

A generalization of the parent-child relationship is the ancestor-descendant relationship.
Tree Terminology (cont.)

A tree consists of a collection of elements or nodes, with each node linked to its successors.

A subtree of a node is a tree whose root is a child of that node.
A tree consists of a collection of elements or nodes, with each node linked to its successors.

A subtree of a node is a tree whose root is a child of that node.
A tree consists of a collection of elements or nodes, with each node linked to its successors.

A subtree of a node is a tree whose root is a child of that node.
Tree Terminology (cont.)

A tree consists of a collection of elements or nodes, with each node linked to its successors.

The level of a node is determined by its distance from the root.
Tree Terminology (cont.)

A tree consists of a collection of elements or nodes, with each node linked to its successors.

The level of a node is its distance from the root plus 1.
Tree Terminology (cont.)

A tree consists of a collection of elements or nodes, with each node linked to its successors.

The level of a node is defined recursively.
A tree consists of a collection of elements or nodes, with each node linked to its successors.

- If node $n$ is the root of tree $T$, its level is 1.
- If node $n$ is not the root of tree $T$, its level is $1 +$ the level of its parent.
A tree consists of a collection of elements or nodes, with each node linked to its successors.

**Tree Terminology (cont.)**

The *height of a tree* is the number of nodes in the longest path from the root node to a leaf node.
A tree consists of a collection of elements or nodes, with each node linked to its successors.

The height of a tree is the number of nodes in the longest path from the root node to a leaf node.

The height of this tree is 3.
Binary Trees

- In a binary tree, each node has two subtrees
- A set of nodes $T$ is a binary tree if either of the following is true:
  - $T$ is empty
  - If $T$ is not empty, it has a root node $r$ with 0, 1, or 2 nonempty binary subtrees whose roots are connected to $r$ by a branch
- ($T_L = \text{left subtree}; \ T_R = \text{right subtree}$)
Binary Trees (cont.)

- $T_L, T_R$, or both can be empty trees
- From now on, we will consistently use a NULL pointer to represent an empty subtree
Expression Tree

- Each node contains an operator or an operand
- Operands are stored in leaf nodes
- Parentheses are not stored in the tree because the tree structure dictates the order of operand evaluation
- Operators in nodes at higher tree levels are evaluated after operators in nodes at lower tree levels

\[(x + y) \ast \left(\frac{a + b}{c}\right)\]
A Huffman tree represents Huffman codes for characters that might appear in a text file.

As opposed to ASCII or Unicode, Huffman code uses different numbers of bits to encode letters; more common characters use fewer bits.

Many programs that compress files use Huffman codes.
To form a code, traverse the tree from the root to the chosen character, appending 0 if you branch left, and 1 if you branch right.
Examples:

d : 10110

e : 010
Binary Search Tree

- Binary search trees
  - All elements in the left subtree precede those in the right subtree
- A formal definition:

A set of nodes $T$ is a binary search tree if either of the following is true:

- $T$ is empty
- If $T$ is not empty, its root node has two subtrees, $T_L$ and $T_R$, such that $T_L$ and $T_R$ are binary search trees and the value in the root node of $T$ is greater than all values in $T_L$ and is less than all values in $T_R$
Recursive Algorithm for Searching a Binary Tree

1. if the tree is empty

2. Return null (target is not found)

else if the target matches the root node's data

3. Return the data stored at the root node

else if the target is less than the root node's data

4. Return the result of searching the left subtree of the root

else

5. Return the result of searching the right subtree of the root
Binary Search Tree (cont.)

- When searching a BST, a typical probe eliminates half the elements in the tree, so if the tree is relatively balanced, searching can be $O(\log n)$.
- In the worst case, searching is $O(n)$. 
The elements in a binary search tree never need to be sorted because they always satisfy the required order relationships.

When new elements are inserted (or removed) properly, the binary search tree maintains its order.

In contrast, a sorted array must be expanded whenever new elements are added, and compacted whenever elements are removed—expanding and contracting are both $O(n)$. 
A full binary tree is a binary tree where all internal nodes have exactly 2 children.

Note that the number of leaf nodes is one more than the number of internal nodes.
A complete binary tree of height $h$ is filled up to depth $h - 1$ and, at depth $h$, any unfilled nodes are on the right.

A node is filled if it has a value stored in it.
More formally, a binary tree of height $h$ is complete if:

- All nodes at depth $h - 2$ and above have two children
- When a node at depth $h - 1$ has children, all nodes to the left of it have two children
- If a node at depth $h - 1$ has one child, it is a left child
We do not explore general trees in this chapter, but nodes of a general tree can have any number of subtrees.
A general tree can be represented using a binary tree.

The left branch of a node is the oldest child, and each right branch is connected to the next younger sibling (if any).
Tree Traversals

Section 8.2
Tree Traversals

- Often we want to determine the nodes of a tree and their relationship
  - We can do this by walking through the tree in a prescribed order and *visiting* the nodes as they are encountered
  - This process is called *tree traversal*

- Three common kinds of tree traversal
  - *Inorder*
  - *Preorder*
  - *Postorder*
Tree Traversals (cont.)

- **Preorder**: visit root node, traverse $T_L$, traverse $T_R$
- **Inorder**: traverse $T_L$, visit root node, traverse $T_R$
- **Postorder**: traverse $T_L$, traverse $T_R$, visit root node

<table>
<thead>
<tr>
<th>Algorithm for Preorder Traversal</th>
<th>Algorithm for Inorder Traversal</th>
<th>Algorithm for Postorder Traversal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. if the tree is empty</td>
<td>1. if the tree is empty</td>
<td>1. if the tree is empty</td>
</tr>
<tr>
<td>2. Return.</td>
<td>2. Return.</td>
<td>2. Return.</td>
</tr>
<tr>
<td>else</td>
<td>else</td>
<td>else</td>
</tr>
<tr>
<td>3. Visit the root.</td>
<td>3. Inorder traverse the left subtree.</td>
<td>3. Postorder traverse the left subtree.</td>
</tr>
<tr>
<td>4. Preorder traverse the left subtree.</td>
<td>4. Visit the root.</td>
<td>4. Postorder traverse the right subtree.</td>
</tr>
<tr>
<td>5. Preorder traverse the right subtree.</td>
<td>5. Inorder traverse the right subtree.</td>
<td>5. Visit the root.</td>
</tr>
</tbody>
</table>
Visualizing Tree Traversals

- You can visualize a tree traversal by imagining a mouse that walks along the edge of the tree.
- If the mouse always keeps the tree to the left, it will trace a route known as the *Euler tour*.
- The Euler tour is the path traced in blue in the figure on the right.
Visualizing Tree Traversals (cont.)

- A Euler tour (blue path) is a preorder traversal
- The sequence in this example is \( a \ b \ d \ g \ e \ h \ c \ f \ i \ j \)
- The mouse visits each node before traversing its subtrees (shown by the downward pointing arrows)
If we record a node as the mouse returns from traversing its left subtree (shown by horizontal black arrows in the figure) we get an inorder traversal.

The sequence is 
\[ d \ g \ b \ h \ e \ a \ i \ f \ j \ c \]
If we record each node as the mouse last encounters it, we get a postorder traversal (shown by the upward pointing arrows).

The sequence is: g d h e b i j f c a
Traversals of Binary Search Trees
and Expression Trees

- An inorder traversal of a binary search tree results in the nodes being visited in sequence by increasing data value.

canine, cat, dog, wolf
Traversals of Binary Search Trees and Expression Trees (cont.)

- An inorder traversal of an expression tree results in the sequence
  \[ x + y * a + b / c \]

- If we insert parentheses where they belong, we get the infix form:
  \[ (x + y) * ((a + b) / c) \]
A postorder traversal of an expression tree results in the sequence
x y + a b + c / *
This is the postfix or reverse polish form of the expression
Operators follow operands
A preorder traversal of an expression tree results in the sequence
\[ * \ + \ x \ y \ / \ + \ a \ b \ c \]
This is the *prefix* form of the expression
Operators precede operands
Binary Search Trees

Section 8.4
Overview of a Binary Search Tree

- Recall the definition of a binary search tree:
  A set of nodes $T$ is a binary search tree if either of the following is true
  - $T$ is empty
  - If $T$ is not empty, its root node has 0, 1, or 2 nonempty subtrees. Its left subtree, $T_L$, is a binary search tree that contains all values less than the value in the root of $T$; its right subtree, $T_R$, is a binary search tree that contains all values greater than the value in the root of $T$

- We will assume that all entries saved in a binary search tree are unique (no duplicates)
Overview of a Binary Search Tree
(cont.)
Recursive Algorithm for Searching a Binary Search Tree

1. if the root is NULL
2. The item is not in the tree; return NULL
3. Compare the value of target with root->data
4. if they are equal
5. The target has been found, return the data at the root
1. else if the target is less than root->data
6. Return the result of searching the left subtree
else
7. Return the result of searching the right subtree
Searching a Binary Tree

Looking for *jill*
Performance

- Search a tree is generally $O(\log n)$
- If a tree is not very full, performance will be worse
- Searching a tree with only right subtrees, for example, is $O(n)$
# The Binary Search Tree Class

<table>
<thead>
<tr>
<th>Function</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bool insert(const Item_Type&amp; item)</code></td>
<td>Inserts an item into the tree. Returns <code>true</code> if the item was inserted, <code>false</code> if the item was already in the tree.</td>
</tr>
<tr>
<td><code>bool erase(const Item_Type&amp; item)</code></td>
<td>Remove an item from the tree. Returns <code>true</code> if the item was removed, <code>false</code> if the item was not in the tree.</td>
</tr>
<tr>
<td><code>const Item_Type* find(const Item_Type&amp; target) const</code></td>
<td>Return a pointer to an item in the tree, or NULL if the item is not in the tree.</td>
</tr>
</tbody>
</table>
The **Binary_Search_Tree** Class (cont.)
Insertion into a Binary Search Tree

Recursive Algorithm for Insertion in a Binary Search Tree

1. if the root is NULL
2. Replace empty tree with a new tree with the item at the root and return true.
3. else if the item is equal to root->data
4. The item is already in the tree; return false.
5. else if the item is less than root->data
6. Recursively insert the item in the left subtree.
7. else
8. Recursively insert the item in the right subtree.
Insertion into a Binary Search Tree
(cont.)

Insert *jill*
Implementing the `insert` Functions

```cpp
template<typename Item_Type>
bool Binary_Bank<Item_Type>::insert(const Item_Type& item) {
  return insert(this->root, item);
}

template<typename Item_Type>
bool Binary_Bank<Item_Type>::insert(BTNode<Item_Type>* local_root, const Item_Type& item) {
  if (local_root == NULL) {
    local_root = new BTNode<Item_Type>(item);
    return true;
  } else {
    if (item < local_root->data)
      return insert(local_root->left, item);
    else if (local_root->data < item)
      return insert(local_root->right, item);
    else {
      return false;
    }
  }
}
```
Removal from a Binary Search Tree

- If the item to be removed is a leaf node, then its parent’s reference to it is set to NULL.
- If the item to be removed has only one child, then the grandparent references the remaining child instead of the child’s parent (the node we want to remove).
Removal from a Binary Search Tree (cont.)

Remove is
If the item to be removed has two children, replace it with the largest item in its left subtree – it *inorder* predecessor.
Removing from a Binary Search Tree (cont.)

Remove rat
Recursive Algorithm for Removal from a Binary Search Tree

1. if the root is NULL
2. The item is not in tree – return NULL.
3. Compare the item to the data at the local root.
4. if the item is less than the data at the local root
5. Return the result of deleting from the left subtree.
6. else if the item is greater than the local root
7. Return the result of deleting from the right subtree.
8. else  // The item is in the local root
9. if the local root has no children
10. Set the parent of the local root to reference NULL.
11. else if the local root has one child
12. Set the parent of the local root to reference that child.
13. else  // Find the inorder predecessor
14. if the left child has no right child,
    it is the inorder predecessor
15. Set the parent of the local root to reference the left child.
16. else
17. Find the rightmost node in the right subtree of the left child.
18. Copy its data into the local root’s data and remove it by setting its parent to reference its left child.
Implementing the \texttt{erase} Functions

\begin{verbatim}
template<typename Item_Type>
bool Binary_Search_Tree<Item_Type>::erase(
    const Item_Type& item) {
    return erase(this->root, item);
}

template<typename Item_Type>
bool Binary_Search_Tree<Item_Type>::erase(
    BTNode<Item_Type>*& local_root,
    const Item_Type& item) {
    if (local_root == NULL) {
        return false;
    } else {
        if (item < local_root->data)
            return erase(local_root->left, item);

    }
\end{verbatim}
else if (local_root->data < item)
    return erase(local_root->right, item);
else {  // Found item
    BTNNode<Item_Type>* old_root = local_root;
    if (local_root->left == NULL) {
        local_root = local_root->right;
    } else if (local_root->right == NULL) {
        local_root = local_root->left;
    } else {
        replace_parent(old_root, old_root->left);
    }
    delete old_root;
    return true;
}
The replace_parent Function
The `replace_parent` Function (cont.)

template<typename Item_Type>
void Binary_Search_Tree<Item_Type>::replace_parent(
    BTNode<Item_Type>*& old_root,
    BTNode<Item_Type>*& local_root) {
    if (local_root->right != NULL) {
        replace_parent(old_root, local_root->right);
    } else {
        old_root->data = local_root->data;
        old_root = local_root;
        local_root = local_root->left;
    }
}
Heaps

Section 8.5
Heaps and Priority Queues

- A heap is a complete binary tree where the value in each node is greater than all values in the node’s subtrees.
- The heap shown below is called a max heap because the root node stores the largest value.
It is also possible to build and use a *min heap*, in which each node’s value is smaller than the values of its children, and the root node stores the smallest value.
The course text presents algorithms and illustrations for max heaps.

Here we present the corresponding algorithms and illustrations for min heaps.
A heap is a complete binary tree with the following properties:

- The value in the root is greater/smaller than or equal to all items in the tree.
- Every subtree is a heap
Inserting an Item into a Min Heap

**Algorithm for Inserting in a Heap**

1. Insert the new item in the next position at the bottom of the heap.
2. while new item is not at the root and new item is smaller than its parent
3. Swap the new item with its parent, moving the new item up the heap.
Algorithm for Inserting in a Heap

1. Insert the new item in the next position at the bottom of the heap.
2. while new item is not at the root and new item is smaller than its parent
3. Swap the new item with its parent, moving the new item up the heap.
Algorithm for Inserting in a Heap
1. Insert the new item in the next position at the bottom of the heap.
2. while new item is not at the root and new item is smaller than its parent
3. Swap the new item with its parent, moving the new item up the heap.
Inserting an Item into a Min Heap
(cont.)

Algorithm for Inserting in a Heap
1. Insert the new item in the next position at the bottom of the heap.
2. while new item is not at the root and new item is smaller than its parent
3. Swap the new item with its parent, moving the new item up the heap.
Inserting an Item into a Min Heap (cont.)

Algorithm for Inserting in a Heap
1. Insert the new item in the next position at the bottom of the heap.
2. while new item is not at the root and new item is smaller than its parent
3. Swap the new item with its parent, moving the new item up the heap.
Removing an Item from a Min Heap

Algorithm for Removal from a Heap

1. Remove the item in the root node by replacing it with the last item in the heap (LIH).
2. while item LIH has children and item LIH is larger than either of its children
3. Swap item LIH with its smaller child, moving LIH down the heap.
Removing an Item from a Heap (cont.)

Algorithm for Removal from a Heap
1. Remove the item in the root node by replacing it with the last item in the heap (LIH).
2. while item LIH has children and item LIH is larger than either of its children
3. Swap item LIH with its smaller child, moving LIH down the heap.
Removing an Item from a Heap (cont.)

Algorithm for Removal from a Heap

1. Remove the item in the root node by replacing it with the last item in the heap (LIH).
2. while item LIH has children and item LIH is larger than either of its children
3. Swap item LIH with its smaller child, moving LIH down the heap.
Removing an Item from a Heap (cont.)

Algorithm for Removal from a Heap
1. Remove the item in the root node by replacing it with the last item in the heap (LIH).
2. While item LIH has children and item LIH is larger than either of its children
3. Swap item LIH with its smaller child, moving LIH down the heap.
Removing an Item from a Heap (cont.)

Algorithm for Removal from a Heap
1. Remove the item in the root node by replacing it with the last item in the heap (LIH).
2. while item LIH has children and item LIH is larger than either of its children
3. Swap item LIH with its smaller child, moving LIH down the heap.
Removing an Item from a Heap (cont.)

Algorithm for Removal from a Heap
1. Remove the item in the root node by replacing it with the last item in the heap (LIH).
2. While item LIH has children and item LIH is larger than either of its children.
3. Swap item LIH with its smaller child, moving LIH down the heap.
Implementing a Min Heap

- Because a heap is a complete binary tree, it can be implemented efficiently using an array rather than a linked data structure.
Implementing a Min Heap (cont.)

```
0 1 2 7 8 9 10 11 12 3 4 5 6
```

```
8
18
29
20 28 39 66
37 26 76 32 74 89
```

```
0 1 2 3 4 5 6 7 8 9 10 11 12
```

```
8 18 29 20 28 39 66 37 26 76 32 74 89
```
Implementing a Min Heap (cont.)

For a node at position $p$,

L. child position: $2p + 1$
R. child position: $2p + 2$
Implementing a Min Heap (cont.)

For a node at position $p$,

- **L. child position:** $2p + 1$
- **R. child position:** $2p + 2$

```
  8
 /   \
18   29
 /     \
20   28   39   66
 /     /     /     \
37  26  76  32   74  89
```

```
  0  1  2  3  4  5  6  7  8  9  10  11  12
 8 18 29 20 28 39 66 37 26 76 32 74 89
```

- Parent
- L. Child
- R. Child
Implementing a Min Heap (cont.)

For a node at position $p$,

L. child position:  $2p + 1$
R. child position:  $2p + 2$
Implementing a Min Heap (cont.)

For a node at position $p$,

L. child position: $2p + 1$
R. child position: $2p + 2$
Implementing a Min Heap (cont.)

For a node at position $p$,
- L. child position: $2p + 1$
- R. child position: $2p + 2$
Implementing a Min Heap (cont.)

A node at position \( c \) can find its parent at \( (c - 1)/2 \).
Inserting into a Min Heap
Implemented as a `Vector`

1. Insert the new element at the end of the `vector` and set `child` to `table.size() - 1`
Inserting into a Min Heap Implemented as a Vector (cont.)

1. Insert the new element at the end of the vector and set child to `table.size() - 1`
Inserting into a Min Heap Implemented as a Vector (cont.)

2. Set parent to (child - 1) / 2
Inserting into a Min Heap Implemented as a Vector (cont.)

3. while (parent >= 0 and table[parent] > table[child])
4. Swap table[parent] and table[child]
5. Set child equal to parent
6. Set parent equal to (child-1)/2
Inserting into a Min Heap Implemented as a Vector (cont.)

3. while (parent >= 0 and table[parent] > table[child])
4. Swap table[parent] and table[child]
5. Set child equal to parent
6. Set parent equal to (child-1)/2
Inserting into a Min Heap Implemented as a Vector (cont.)

3. while (parent >= 0 and table[parent] > table[child])
4. Swap table[parent] and table[child]
5. Set child equal to parent
6. Set parent equal to (child-1)/2
Inserting into a Min Heap Implemented as a Vector (cont.)

3. while (parent >= 0 and table[parent] > table[child])
4. Swap table[parent] and table[child]
5. Set child equal to parent
6. Set parent equal to (child-1)/2
Inserting into a Min Heap Implemented as a Vector (cont.)

3. while (parent >= 0 and table[parent] > table[child])
4. Swap table[parent] and table[child]
5. Set child equal to parent
6. Set parent equal to (child-1)/2
Inserting into a Min Heap Implemented as a Vector (cont.)

3. while (parent >= 0 and table[parent] > table[child])
4. Swap table[parent] and table[child]
5. Set child equal to parent
6. Set parent equal to (child-1)/2

```
6 18 8 20 28 39 29 37 26 76 32 74 89
```

- Child
- Parent
Inserting into a Min Heap Implemented as a Vector (cont.)

3. while (parent >= 0 and table[parent] > table[child])
4. Swap table[parent] and table[child]
5. Set child equal to parent
6. Set parent equal to (child-1)/2
Inserting into a Min Heap Implemented as a Vector (cont.)

3. while (parent >= 0 and table[parent] > table[child])
4. Swap table[parent] and table[child]
5. Set child equal to parent
6. Set parent equal to (child-1)/2
Inserting into a Min Heap Implemented as a Vector (cont.)

3. while (parent >= 0 and table[parent] > table[child])
4. Swap table[parent] and table[child]
5. Set child equal to parent
6. Set parent equal to (child-1)/2
Removal from a Min Heap

Implemented as a vector

1. Remove the last element (i.e., the one at size() – 1) and set the item at 0 to this value.
2. Set parent to 0.
3. while (true)
   4. Set left_child to (2 * parent) + 1 and right_child to left_child + 1
   5. if leftChild >= table.size()
   7. Assume min_child (the smaller child) is left_child
   8. if right_child < table.size() and table[left_child] > table[right_child]
      9. Set min_child to right_child
   10. if table[parent] > table[min_child]
      11. Swap table[parent] and table[min_child]
      12. Set parent to min_child
      else
          Break out of loop.
Performance of the Heap

- The removal algorithm traces a path from the root to a leaf.
- The insertion algorithm traces a path from a leaf to the root.
- This requires at most $h$ steps where $h$ is the height of the tree.
- The largest full tree of height $h$ has $2^h - 1$ nodes.
- The smallest complete tree of height $h$ has $2^{(h-1)}$ nodes.
- Both insert and remove are $O(\log n)$ where $n$ is the number of items in the heap.
The Queue Abstract Data Type

Section 6.1
Queues

- The queue, like the stack, is a widely used data structure.

- A queue differs from a stack in one important way:
  - A stack is LIFO list – *Last-In, First-Out*
  - While a queue is FIFO list, *First-In, First-Out*
The Queue Abstract Data Type

Section 6.1
Queue Abstract Data Type

- A queue can be visualized as a line of customers waiting for service.
- The next person to be served is the one who has waited the longest.
- New elements are placed at the end of the line.
A Queue of Customers

- To the left is a queue of three customers waiting to buy concert tickets

<table>
<thead>
<tr>
<th>Ticket agent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thome</td>
</tr>
<tr>
<td>Abreu</td>
</tr>
<tr>
<td>Jones</td>
</tr>
</tbody>
</table>
A Queue of Customers

- To the left is a queue of three customers waiting to buy concert tickets

Thome
Abreu
Jones
A Queue of Customers

- To the left is a queue of three customers waiting to buy concert tickets.

<table>
<thead>
<tr>
<th>Thome</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abreu</td>
</tr>
<tr>
<td>Jones</td>
</tr>
</tbody>
</table>

Jones is the most recent arrival.
A Queue of Customers

- To the left is a queue of three customers waiting to buy concert tickets.

- Thome will be the first customer removed from the queue (and able to buy tickets) when a ticket agent becomes available.
A Queue of Customers

- To the left is a queue of three customers waiting to buy concert tickets.

  Thome

  Abreu

  Jones

Ticket agent

Abreu will then become the first one in the queue.
A Queue of Customers

- To the left is a queue of three customers waiting to buy concert tickets.

| Thome | Abreu | Jones |

Any new customers will be inserted in the queue after Jones.
Print Queue

- Operating systems use queues to
  - keep track of tasks waiting for a scarce resource
  - ensure that the tasks are carried out in the order they were generated

- Print queue: printing typically is much slower than the process of selecting pages to print, so a queue is used
Specification for a Queue Interface

Because only the front element of a queue is visible, the operations performed by a queue are few in number.

- We need to be able to retrieve the front element, remove the front element, push a new element onto the queue, and test for an empty queue.

- The functions above are all defined in the header file for the STL container queue, `<queue>`.

<table>
<thead>
<tr>
<th>Function</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool empty() const</td>
<td>Returns <code>true</code> if the queue is empty; otherwise returns <code>false</code>.</td>
</tr>
<tr>
<td>Item_Type&amp; front();</td>
<td>Returns the object at the front of the queue without removing it.</td>
</tr>
<tr>
<td>const Item_Type&amp; front() const</td>
<td></td>
</tr>
<tr>
<td>void pop()</td>
<td>Removes the object at the front of the queue.</td>
</tr>
<tr>
<td>void push(const Item_Type&amp;)</td>
<td>Pushes an item onto the rear of the queue.</td>
</tr>
<tr>
<td>size_t size() const</td>
<td>Returns the number of items in the queue.</td>
</tr>
</tbody>
</table>
Specifying for a Queue Interface (cont.)

- For queue names in (a), the value of names.empty() is false.

- The statement
  ```
  string first = names.front();
  ```
  stores "Jonathan" in first without changing names.

![Image](image.png)
The statement
names.pop();
removes "Jonathan" from names. The queue names now contains four elements and is shown in (b)
Specification for a Queue Interface (cont.)

- The statement `names.push("Eliana");`

- adds "Eliana" to the end of the queue; the queue names now contains five elements and is shown in (c)
Priority Queues

Section 8.5
Priority Queues

- The heap is used to implement a special kind of queue called a priority queue
- Sometimes a FIFO queue may not be the best way to implement a waiting line
- A priority queue is a data structure in which only the highest-priority item is accessible
In a print queue, sometimes it is more appropriate to print a short document that arrived after a very long document.

A priority queue is a data structure in which only the highest-priority item is accessible (as opposed to the first item entered).
Insertion into a Priority Queue

<table>
<thead>
<tr>
<th>Pages</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&quot;web page 1&quot;</td>
</tr>
<tr>
<td>4</td>
<td>&quot;history paper&quot;</td>
</tr>
</tbody>
</table>

After inserting document with 3 pages:

<table>
<thead>
<tr>
<th>Pages</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&quot;web page 1&quot;</td>
</tr>
<tr>
<td>3</td>
<td>&quot;Lab1&quot;</td>
</tr>
<tr>
<td>4</td>
<td>&quot;history paper&quot;</td>
</tr>
</tbody>
</table>

After inserting document with 1 page:

<table>
<thead>
<tr>
<th>Pages</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&quot;web page 1&quot;</td>
</tr>
<tr>
<td>1</td>
<td>&quot;receipt&quot;</td>
</tr>
<tr>
<td>3</td>
<td>&quot;Lab1&quot;</td>
</tr>
<tr>
<td>4</td>
<td>&quot;history paper&quot;</td>
</tr>
</tbody>
</table>
C++ provides a priority_queue class that uses the same interface as the queue given in Chapter 6.

The differences are in the specification for the `top` and `pop` functions—these are defined to return a largest item in the queue rather than an oldest item.

<table>
<thead>
<tr>
<th>Function</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void push(const Item_Type&amp; item)</code></td>
<td>Inserts an item into the queue.</td>
</tr>
<tr>
<td><code>void pop()</code></td>
<td>Removes the largest entry if the queue is not empty. If the queue is empty, a run-time error may occur.</td>
</tr>
<tr>
<td><code>const Item_Type&amp; top() const</code></td>
<td>Returns the largest entry without removing it. If the queue is empty, a run-time error may occur.</td>
</tr>
<tr>
<td><code>size_t size() const</code></td>
<td>Returns the number of items in the priority queue.</td>
</tr>
<tr>
<td><code>bool empty() const</code></td>
<td>Returns <code>true</code> if the queue is empty, <code>false</code> otherwise.</td>
</tr>
</tbody>
</table>