Binary Trees, Binary Search Trees, and Heaps

Victor Milenkovic

Department of Computer Science
University of Miami

CSC220 Programming II – Spring 2020

Review: What is a list?

Review: What is a list?

- A list is a collection of elements.

Review: What is a list?

- A list is a collection of elements.
  - Each element has at most one successor.

Review: What is a list?

- A list is a collection of elements.
  - Each element has at most one successor.
  - Each element is the successor of at most one element, its *predecessor*.

Review: What is a list?

- A list is a collection of elements.
  - Each element has at most one successor.
  - Each element is the successor of at most one element, its *predecessor*.
  - The first element has no predecessor.

Review: What is a list?

- A list is a collection of elements.
  - Each element has at most one successor.
  - Each element is the successor of at most one element, its *predecessor*.
  - The first element has no predecessor.
  - The last element has no successor.

Review: What is a list?

- A list is a collection of elements.
  - Each element has at most one successor.
  - Each element is the successor of at most one element, its *predecessor*.
  - The first element has no predecessor.
  - The last element has no successor.
- A list is often written vertically.

Review: What is a list?

- A list is a collection of elements.
  - Each element has at most one successor.
  - Each element is the successor of at most one element, its *predecessor*.
  - The first element has no predecessor.
  - The last element has no successor.
- A list is often written vertically.
  - Like a grocery list.

Review: What is a list?

- A list is a collection of elements.
  - Each element has at most one successor.
  - Each element is the successor of at most one element, its *predecessor*.
  - The first element has no predecessor.
  - The last element has no successor.
- A list is often written vertically.
  - Like a grocery list.
  - First at the top.

Review: What is a list?

- A list is a collection of elements.
  - Each element has at most one successor.
  - Each element is the successor of at most one element, its *predecessor*.
  - The first element has no predecessor.
  - The last element has no successor.
- A list is often written vertically.
  - Like a grocery list.
  - First at the top.
  - Last at the bottom.

Review: What is a list?

- A list is a collection of elements.
  - Each element has at most one successor.
  - Each element is the successor of at most one element, its *predecessor*.
  - The first element has no predecessor.
  - The last element has no successor.
- A list is often written vertically.
  - Like a grocery list.
  - First at the top.
  - Last at the bottom.
    - eggs

Review: What is a list?

- A list is a collection of elements.
  - Each element has at most one successor.
  - Each element is the successor of at most one element, its *predecessor*.
  - The first element has no predecessor.
  - The last element has no successor.
- A list is often written vertically.
  - Like a grocery list.
  - First at the top.
  - Last at the bottom.
    - eggs
    - milk

Review: What is a list?

- A list is a collection of elements.
  - Each element has at most one successor.
  - Each element is the successor of at most one element, its *predecessor*.
  - The first element has no predecessor.
  - The last element has no successor.
- A list is often written vertically.
  - Like a grocery list.
  - First at the top.
  - Last at the bottom.
    - eggs
    - milk
    - bread

Review: What is a list?

- A list is a collection of elements.
  - Each element has at most one successor.
  - Each element is the successor of at most one element, its *predecessor*.
  - The first element has no predecessor.
  - The last element has no successor.
- A list is often written vertically.
  - Like a grocery list.
  - First at the top.
  - Last at the bottom.
    - eggs
    - milk
    - bread
    - apples

Review: What is a list?

- A list is a collection of elements.
  - Each element has at most one successor.
  - Each element is the successor of at most one element, its *predecessor*.
  - The first element has no predecessor.
  - The last element has no successor.
- A list is often written vertically.
  - Like a grocery list.
  - First at the top.
  - Last at the bottom.
    - eggs
    - milk
    - bread
    - apples
    - pasta

Review: What is a list?

- A list is a collection of elements.
  - Each element has at most one successor.
  - Each element is the successor of at most one element, its *predecessor*.
  - The first element has no predecessor.
  - The last element has no successor.
- A list is often written vertically.
  - Like a grocery list.
  - First at the top.
  - Last at the bottom.
    - eggs
    - milk
    - bread
    - apples
    - pasta
    - cheese

Review: representations of list

Review: representations of list

- Linked representation

Review: representations of list

- Linked representation
  - **first** variable points to entry with first element.

Review: representations of list

- Linked representation
  - **first** variable points to entry with first element.
  - **last** variable points to entry with last element.

Review: representations of list

- Linked representation
  - **first** variable points to entry with first element.
  - **last** variable points to entry with last element.
  - Successor of **entry** is **entry.next**.

Review: representations of list

- Linked representation
  - **first** variable points to entry with first element.
  - **last** variable points to entry with last element.
  - Successor of **entry** is **entry.next**.
  - Optional **entry.previous** is predecessor.

Review: representations of list

- Linked representation
  - **first** variable points to entry with first element.
  - **last** variable points to entry with last element.
  - Successor of **entry** is **entry.next**.
  - Optional **entry.previous** is predecessor.
- Array representation

Review: representations of list

- Linked representation
  - **first** variable points to entry with first element.
  - **last** variable points to entry with last element.
  - Successor of **entry** is **entry.next**.
  - Optional **entry.previous** is predecessor.
- Array representation
  - **theElements** is array of elements.

Review: representations of list

- Linked representation
  - **first** variable points to entry with first element.
  - **last** variable points to entry with last element.
  - Successor of **entry** is **entry.next**.
  - Optional **entry.previous** is predecessor.
- Array representation
  - **theElements** is array of elements.
  - **size** is number of elements.

Review: representations of list

- Linked representation
  - **first** variable points to entry with first element.
  - **last** variable points to entry with last element.
  - Successor of **entry** is **entry.next**.
  - Optional **entry.previous** is predecessor.
- Array representation
  - **theElements** is array of elements.
  - **size** is number of elements.
  - First element is **theElements[0]**.

Review: representations of list

- Linked representation
  - **first** variable points to entry with first element.
  - **last** variable points to entry with last element.
  - Successor of **entry** is **entry.next**.
  - Optional **entry.previous** is predecessor.
- Array representation
  - **theElements** is array of elements.
  - **size** is number of elements.
  - First element is **theElements[0]**.
  - Last element is **theElements[size**-1**]**.

Review: representations of list

- Linked representation
  - **first** variable points to entry with first element.
  - **last** variable points to entry with last element.
  - Successor of **entry** is **entry.next**.
  - Optional **entry.previous** is predecessor.
- Array representation
  - **theElements** is array of elements.
  - **size** is number of elements.
  - First element is **theElements[0]**.
  - Last element is **theElements[size−1]**.
  - For $0 \leq i < \text{size} - 1$, successor of **theElements[i]** is **theElements[$i+1$]**.

Review: representations of list

- Linked representation
  - **first** variable points to entry with first element.
  - **last** variable points to entry with last element.
  - Successor of **entry** is **entry.next**.
  - Optional **entry.previous** is predecessor.
- Array representation
  - **theElements** is array of elements.
  - **size** is number of elements.
  - First element is **theElements[0]**.
  - Last element is **theElements[size**$-$**1]**.
  - For $0 \leq i < \text{size} - 1$, successor of **theElements[**i**]** is **theElements[**$i + 1$**]**.
  - For $0 < j < \text{size}$, predecessor of **theElements[**j**]** is **theElements[**$j - 1$**]**.

Review: representations of list

- Linked representation
  - **first** variable points to entry with first element.
  - **last** variable points to entry with last element.
  - Successor of **entry** is **entry.next**.
  - Optional **entry.previous** is predecessor.
- Array representation
  - **theElements** is array of elements.
  - **size** is number of elements.
  - First element is **theElements[0]**.
  - Last element is **theElements[size$-$1]**.
  - For $0 \leq i <$ size $- 1$, successor of **theElements[*i*]** is **theElements[*i* $+ 1$]**.
  - For $0 < j <$ size, predecessor of **theElements[*j*]** is **theElements[*j* $- 1$]**.
  - **theElements[*i*]** does not have a successor if

Review: representations of list

- Linked representation
  - **first** variable points to entry with first element.
  - **last** variable points to entry with last element.
  - Successor of **entry** is **entry.next**.
  - Optional **entry.previous** is predecessor.
- Array representation
  - **theElements** is array of elements.
  - **size** is number of elements.
  - First element is **theElements[0]**.
  - Last element is **theElements[size$-$1]**.
  - For $0 \leq i <$ size $- 1$, successor of **theElements[i]** is **theElements[$i+1$]**.
  - For $0 < j <$ size, predecessor of **theElements[j]** is **theElements[$j-1$]**.
  - **theElements[i]** does not have a successor if $i+1 \geq$size.

Review: representations of list

- Linked representation
  - **first** variable points to entry with first element.
  - **last** variable points to entry with last element.
  - Successor of **entry** is **entry.next**.
  - Optional **entry.previous** is predecessor.
- Array representation
  - **theElements** is array of elements.
  - **size** is number of elements.
  - First element is **theElements[0]**.
  - Last element is **theElements[size$-$1]**.
  - For $0 \leq i < \text{size} - 1$, successor of **theElements[i]** is **theElements[$i+1$]**.
  - For $0 < j < \text{size}$, predecessor of **theElements[j]** is **theElements[$j-1$]**.
  - **theElements[i]** does not have a successor if $i + 1 \geq$size.
  - **theElements[j]** does not have a predecessor if

Review: representations of list

- Linked representation
  - **first** variable points to entry with first element.
  - **last** variable points to entry with last element.
  - Successor of **entry** is **entry.next**.
  - Optional **entry.previous** is predecessor.
- Array representation
  - **theElements** is array of elements.
  - **size** is number of elements.
  - First element is **theElements[0]**.
  - Last element is **theElements[size$-$1]**.
  - For $0 \le i <$ size $- 1$, successor of **theElements[i]** is **theElements[$i + 1$]**.
  - For $0 < j <$ size, predecessor of **theElements[j]** is **theElements[$j - 1$]**.
  - **theElements[i]** does not have a successor if $i + 1 \ge$size.
  - **theElements[j]** does not have a predecessor if $j = 0$.

Review: list order

Review: list order

- Possible list orders:

Review: list order

- Possible list orders:
  - unsorted,

Review: list order

- Possible list orders:
  - unsorted,
    - eggs

Review: list order

- Possible list orders:
  - unsorted,
    - eggs
    - milk

Review: list order

- Possible list orders:
  - unsorted,
    - eggs
    - milk
    - bread

Review: list order

- Possible list orders:
  - unsorted,
    - eggs
    - milk
    - bread
    - apples

Review: list order

- Possible list orders:
  - unsorted,
    - eggs
    - milk
    - bread
    - apples
    - pasta

Review: list order

- Possible list orders:
  - unsorted,
    - eggs
    - milk
    - bread
    - apples
    - pasta
    - cheese

Review: list order

- Possible list orders:
  - unsorted,
    - eggs
    - milk
    - bread
    - apples
    - pasta
    - cheese
  - sorted.

Review: list order

- Possible list orders:
  - unsorted,
    - eggs
    - milk
    - bread
    - apples
    - pasta
    - cheese
  - sorted.
    - apples

Review: list order

- Possible list orders:
  - unsorted,
    - eggs
    - milk
    - bread
    - apples
    - pasta
    - cheese
  - sorted.
    - apples
    - bread

Review: list order

- Possible list orders:
  - unsorted,
    - eggs
    - milk
    - bread
    - apples
    - pasta
    - cheese
  - sorted.
    - apples
    - bread
    - cheese

Review: list order

- Possible list orders:
  - unsorted,
    - eggs
    - milk
    - bread
    - apples
    - pasta
    - cheese
  - sorted.
    - apples
    - bread
    - cheese
    - eggs

Review: list order

- Possible list orders:
  - unsorted,
    - eggs
    - milk
    - bread
    - apples
    - pasta
    - cheese
  - sorted.
    - apples
    - bread
    - cheese
    - eggs
    - milk

Review: list order

- Possible list orders:
  - unsorted,
    - eggs
    - milk
    - bread
    - apples
    - pasta
    - cheese
  - sorted.
    - apples
    - bread
    - cheese
    - eggs
    - milk
    - pasta

Tree

Tree

- A *binary tree* is like a list except:

Tree

- A *binary tree* is like a list except:
  - Each element has up to one *left successor* and up to one *right successor*.

Tree

- A *binary tree* is like a list except:
  - Each element has up to one *left successor* and up to one *right successor*.
  - These are called the left and right *children*.

Tree

- A *binary tree* is like a list except:
  - Each element has up to one *left successor* and up to one *right successor*.
  - These are called the left and right *children*.
  - The (up to one) predecessor is called the *parent*.

Tree

- A *binary tree* is like a list except:
  - Each element has up to one *left successor* and up to one *right successor*.
  - These are called the left and right *children*.
  - The (up to one) predecessor is called the *parent*.
  - The element without a predecessor is called the *root*.

Tree

- A *binary tree* is like a list except:
  - Each element has up to one *left successor* and up to one *right successor*.
  - These are called the left and right *children*.
  - The (up to one) predecessor is called the *parent*.
  - The element without a predecessor is called the *root*.
  - Elements without any successors are called *leaves*.

Tree

- A *binary tree* is like a list except:
  - Each element has up to one *left successor* and up to one *right successor*.
  - These are called the left and right *children*.
  - The (up to one) predecessor is called the *parent*.
  - The element without a predecessor is called the *root*.
  - Elements without any successors are called *leaves*.
- Tree diagram:

Tree

- A *binary tree* is like a list except:
  - Each element has up to one *left successor* and up to one *right successor*.
  - These are called the left and right *children*.
  - The (up to one) predecessor is called the *parent*.
  - The element without a predecessor is called the *root*.
  - Elements without any successors are called *leaves*.
- Tree diagram:
  - Root at the top.

Tree

- A *binary tree* is like a list except:
  - Each element has up to one *left successor* and up to one *right successor*.
  - These are called the left and right *children*.
  - The (up to one) predecessor is called the *parent*.
  - The element without a predecessor is called the *root*.
  - Elements without any successors are called *leaves*.
- Tree diagram:
  - Root at the top.
  - Element connect to left and right child diagonally downward.

Tree

- A *binary tree* is like a list except:
  - Each element has up to one *left successor* and up to one *right successor*.
  - These are called the left and right *children*.
  - The (up to one) predecessor is called the *parent*.
  - The element without a predecessor is called the *root*.
  - Elements without any successors are called *leaves*.
- Tree diagram:
  - Root at the top.
  - Element connect to left and right child diagonally downward.
- Other terms:

Tree

- A *binary tree* is like a list except:
  - Each element has up to one *left successor* and up to one *right successor*.
  - These are called the left and right *children*.
  - The (up to one) predecessor is called the *parent*.
  - The element without a predecessor is called the *root*.
  - Elements without any successors are called *leaves*.
- Tree diagram:
  - Root at the top.
  - Element connect to left and right child diagonally downward.
- Other terms:
  - *Depth* of element is the number of steps from a root.

Tree

- A *binary tree* is like a list except:
  - Each element has up to one *left successor* and up to one *right successor*.
  - These are called the left and right *children*.
  - The (up to one) predecessor is called the *parent*.
  - The element without a predecessor is called the *root*.
  - Elements without any successors are called *leaves*.
- Tree diagram:
  - Root at the top.
  - Element connect to left and right child diagonally downward.
- Other terms:
  - *Depth* of element is the number of steps from a root.
  - *Height* of tree is depth of deepest leaf.

Tree

- A *binary tree* is like a list except:
  - Each element has up to one *left successor* and up to one *right successor*.
  - These are called the left and right *children*.
  - The (up to one) predecessor is called the *parent*.
  - The element without a predecessor is called the *root*.
  - Elements without any successors are called *leaves*.
- Tree diagram:
  - Root at the top.
  - Element connect to left and right child diagonally downward.
- Other terms:
  - *Depth* of element is the number of steps from a root.
  - *Height* of tree is depth of deepest leaf.
  - The portion of a tree at or below an element is called its *subtree*.

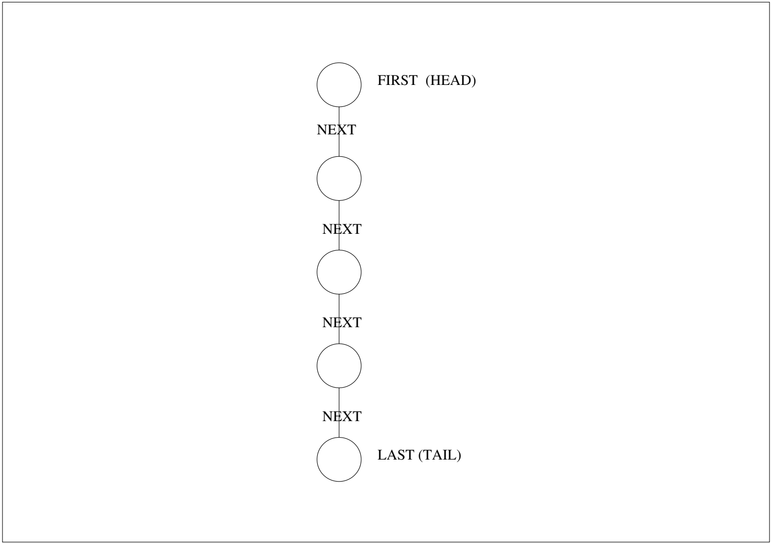
Tree

- A *binary tree* is like a list except:
  - Each element has up to one *left successor* and up to one *right successor*.
  - These are called the left and right *children*.
  - The (up to one) predecessor is called the *parent*.
  - The element without a predecessor is called the *root*.
  - Elements without any successors are called *leaves*.
- Tree diagram:
  - Root at the top.
  - Element connect to left and right child diagonally downward.
- Other terms:
  - *Depth* of element is the number of steps from a root.
  - *Height* of tree is depth of deepest leaf.
  - The portion of a tree at or below an element is called its *subtree*.
  - The subtree of its left child is called its *left subtree*.

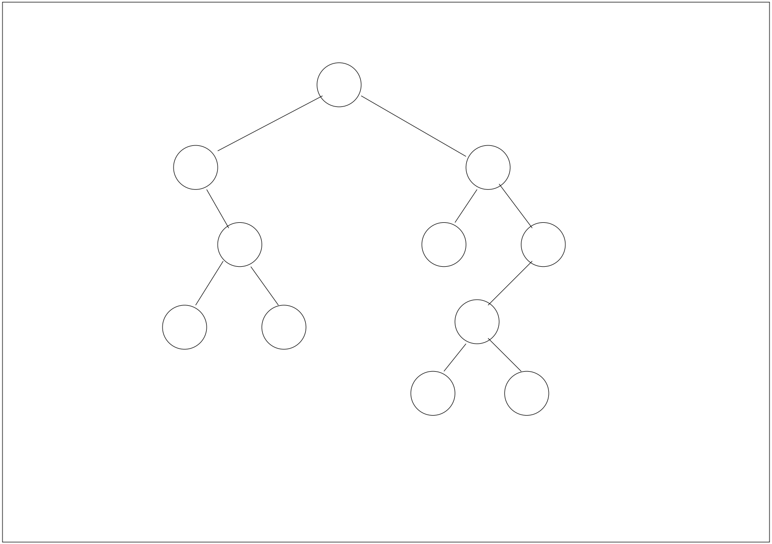
Tree

- A *binary tree* is like a list except:
  - Each element has up to one *left successor* and up to one *right successor*.
  - These are called the left and right *children*.
  - The (up to one) predecessor is called the *parent*.
  - The element without a predecessor is called the *root*.
  - Elements without any successors are called *leaves*.
- Tree diagram:
  - Root at the top.
  - Element connect to left and right child diagonally downward.
- Other terms:
  - *Depth* of element is the number of steps from a root.
  - *Height* of tree is depth of deepest leaf.
  - The portion of a tree at or below an element is called its *subtree*.
  - The subtree of its left child is called its *left subtree*.
  - The subtree of its right child is called its *right subtree*.

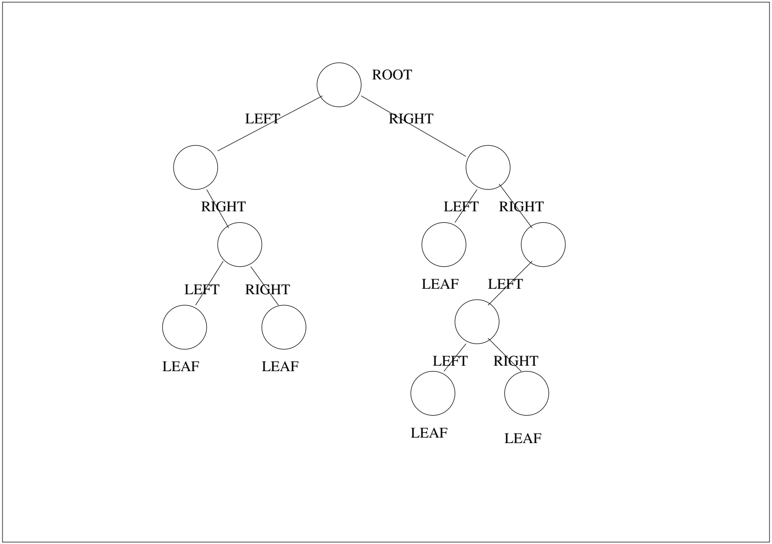
List



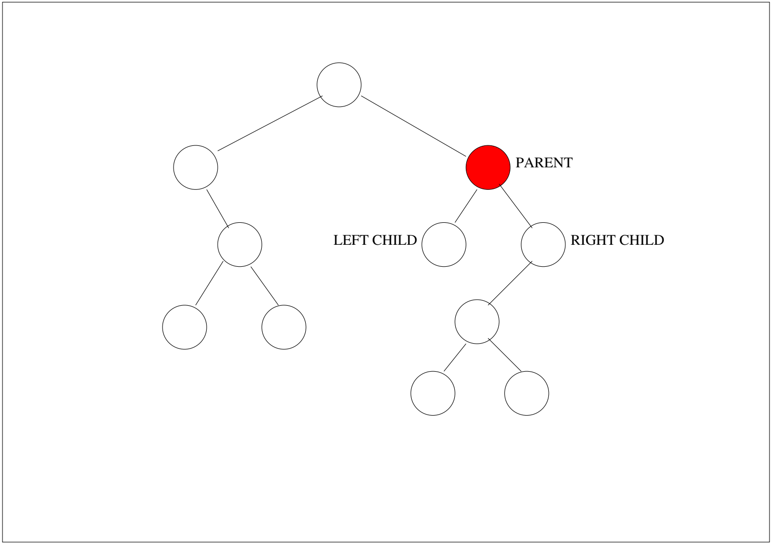
Binary Tree



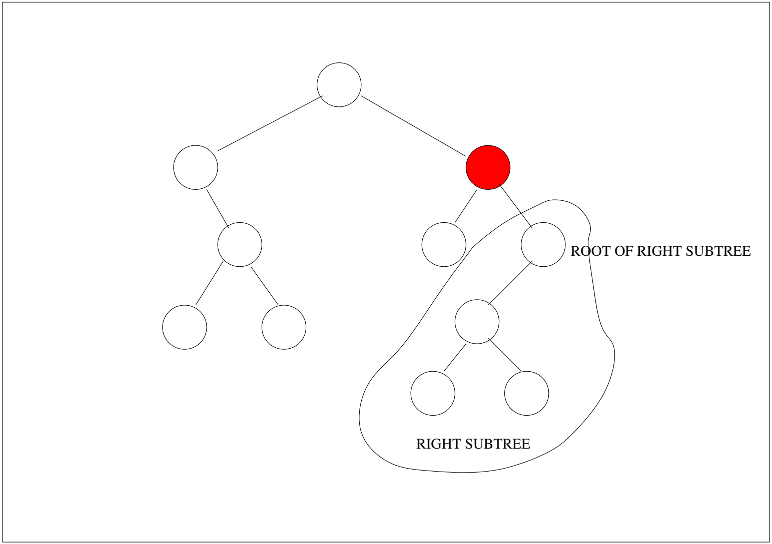
Roots and Leaves



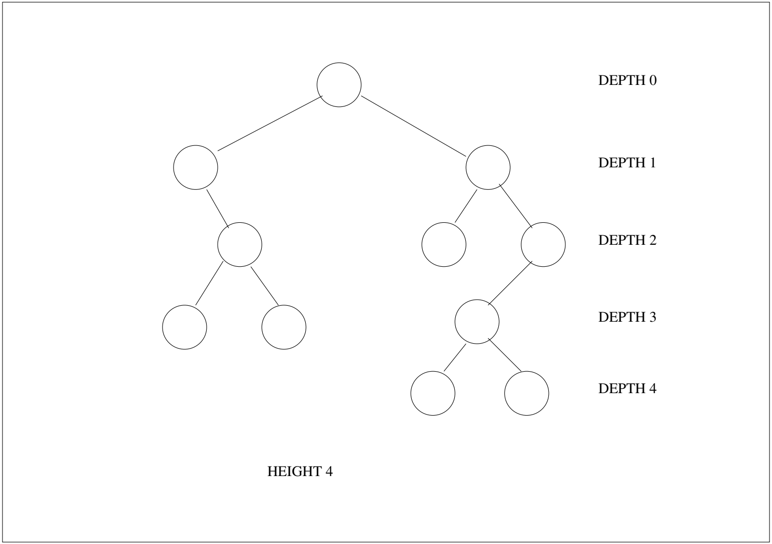
Parent, Left Child, Right Child



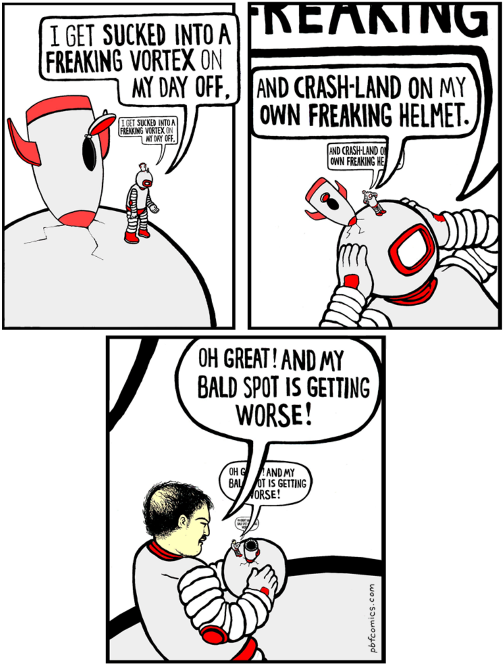
Subtree



Depth of Element and Height of Tree

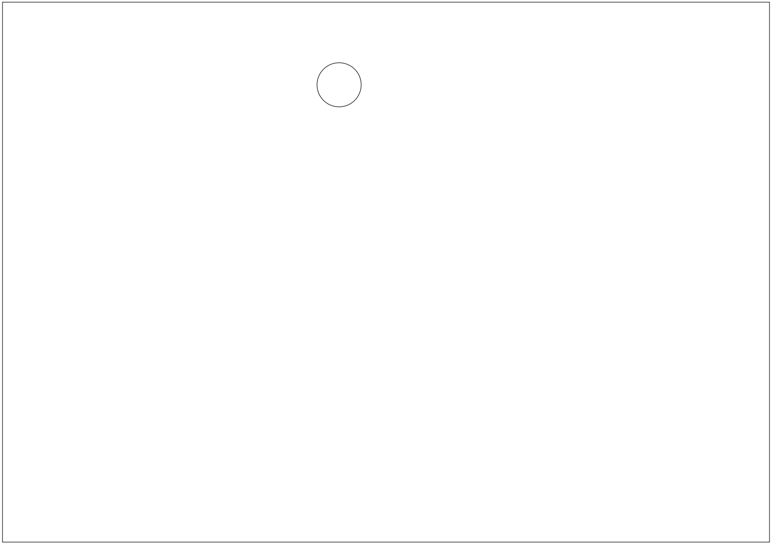


Trees can be defined RECURSIVELY

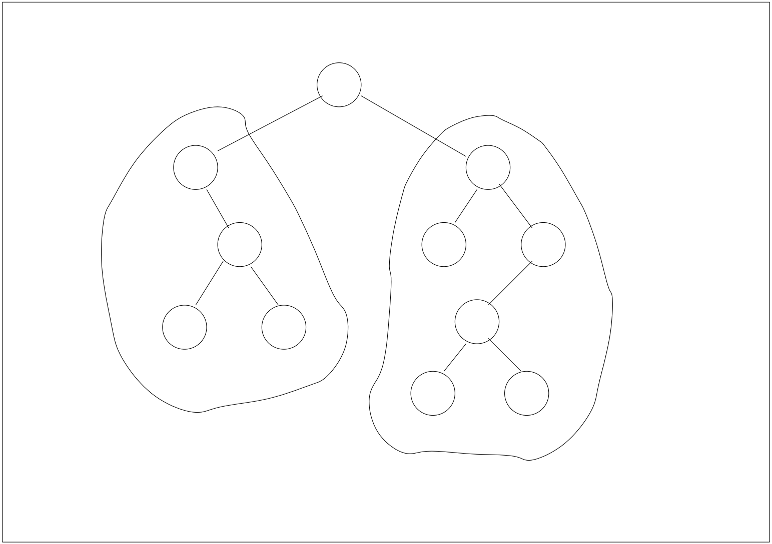


Recursive Definition: Empty

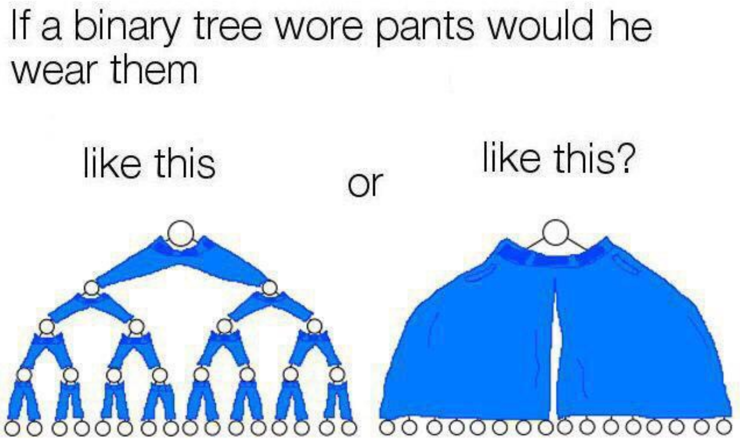
or a single element



with a left and right (sub)tree



Question


If a binary tree wore pants would he wear them
like this
or
like this?

Tree Representations

Tree Representations

- Linked

Tree Representations

- Linked
  - **root** variable points to root entry.

Tree Representations

- Linked
  - **root** variable points to root entry.
  - For each **entry** (starting with **root**),

Tree Representations

- Linked
  - **root** variable points to root entry.
  - For each **entry** (starting with **root**),
  - **entry.left** and **entry.right** point to left and right child of **entry**.

Tree Representations

- Linked
  - **root** variable points to root entry.
  - For each **entry** (starting with **root**),
  - **entry.left** and **entry.right** point to left and right child of **entry**.
  - Either one can be null.

Tree Representations

- Linked
  - **root** variable points to root entry.
  - For each **entry** (starting with **root**),
  - **entry.left** and **entry.right** point to left and right child of **entry**.
  - Either one can be null.
- Array

Tree Representations

- Linked
  - **root** variable points to root entry.
  - For each **entry** (starting with **root**),
  - **entry.left** and **entry.right** point to left and right child of **entry**.
  - Either one can be null.
- Array
  - **theElements** is an array of elements.

Tree Representations

- Linked
  - **root** variable points to root entry.
  - For each **entry** (starting with **root**),
  - **entry.left** and **entry.right** point to left and right child of **entry**.
  - Either one can be null.
- Array
  - **theElements** is an array of elements.
  - **size** indicate that indices 0 to size-1 are valid.

Tree Representations

- Linked
  - **root** variable points to root entry.
  - For each **entry** (starting with **root**),
  - **entry.left** and **entry.right** point to left and right child of **entry**.
  - Either one can be null.
- Array
  - **theElements** is an array of elements.
  - **size** indicate that indices 0 to size-1 are valid.
  - Root is at **theElements[0]**.

Tree Representations

- Linked
  - **root** variable points to root entry.
  - For each **entry** (starting with **root**),
  - **entry.left** and **entry.right** point to left and right child of **entry**.
  - Either one can be null.
- Array
  - **theElements** is an array of elements.
  - **size** indicate that indices 0 to size-1 are valid.
  - Root is at **theElements[0]**.
  - Left child of **theElements[i]** is at **theElements[$2i+1$]**, but only if $2i+1 <$size. Otherwise, there is no left child.

Tree Representations

- Linked
  - **root** variable points to root entry.
  - For each **entry** (starting with **root**),
  - **entry.left** and **entry.right** point to left and right child of **entry**.
  - Either one can be null.
- Array
  - **theElements** is an array of elements.
  - **size** indicate that indices 0 to size-1 are valid.
  - Root is at **theElements[0]**.
  - Left child of **theElements[*i*]** is at **theElements[$2i+1$]**, but only if $2i+1<$size. Otherwise, there is no left child.
  - Right child of **theElements[*i*]** is at **theElements[$2i+2$]**, but only if $2i+2<$size. Otherwise, there is no right child.

Tree Representations

- Linked
  - **root** variable points to root entry.
  - For each **entry** (starting with **root**),
  - **entry.left** and **entry.right** point to left and right child of **entry**.
  - Either one can be null.
- Array
  - **theElements** is an array of elements.
  - **size** indicate that indices 0 to size-1 are valid.
  - Root is at **theElements[0]**.
  - Left child of **theElements[*i*]** is at **theElements[**$2i+1$**]**, but only if $2i+1 <$size. Otherwise, there is no left child.
  - Right child of **theElements[*i*]** is at **theElements[**$2i+2$**]**, but only if $2i+2 <$size. Otherwise, there is no right child.
  - For $0 < j <$size, parent of **theElements[*j*]** is at

Tree Representations

- Linked
  - **root** variable points to root entry.
  - For each **entry** (starting with **root**),
  - **entry.left** and **entry.right** point to left and right child of **entry**.
  - Either one can be null.
- Array
  - **theElements** is an array of elements.
  - **size** indicate that indices 0 to size-1 are valid.
  - Root is at **theElements[0]**.
  - Left child of **theElements[*i*]** is at **theElements[**$2i+1$**]**, but only if $2i+1<$size. Otherwise, there is no left child.
  - Right child of **theElements[*i*]** is at **theElements[**$2i+2$**]**, but only if $2i+2<$size. Otherwise, there is no right child.
  - For $0<j<$size, parent of **theElements[*j*]** is at **theElements[**$(j-1)/2$**]**.

Tree Representations

- Linked
  - **root** variable points to root entry.
  - For each **entry** (starting with **root**),
  - **entry.left** and **entry.right** point to left and right child of **entry**.
  - Either one can be null.
- Array
  - **theElements** is an array of elements.
  - **size** indicate that indices 0 to size-1 are valid.
  - Root is at **theElements[0]**.
  - Left child of **theElements[*i*]** is at **theElements[**$2i+1$**]**, but only if $2i+1 <$size. Otherwise, there is no left child.
  - Right child of **theElements[*i*]** is at **theElements[**$2i+2$**]**, but only if $2i+2 <$size. Otherwise, there is no right child.
  - For $0 < j <$size, parent of **theElements[*j*]** is at **theElements[**$(j-1)/2$**]**.
- Array representation wastes a lot of space unless tree is *complete*.

Tree Representations

- Linked
  - **root** variable points to root entry.
  - For each **entry** (starting with **root**),
  - **entry.left** and **entry.right** point to left and right child of **entry**.
  - Either one can be null.
- Array
  - **theElements** is an array of elements.
  - **size** indicate that indices 0 to size-1 are valid.
  - Root is at **theElements[0]**.
  - Left child of **theElements[i]** is at **theElements[$2i+1$]**, but only if $2i+1 <$size. Otherwise, there is no left child.
  - Right child of **theElements[i]** is at **theElements[$2i+2$]**, but only if $2i+2 <$size. Otherwise, there is no right child.
  - For $0 < j <$size, parent of **theElements[j]** is at **theElements[$(j-1)/2$]**.
- Array representation wastes a lot of space unless tree is *complete*.
  - All levels except the bottom level are full.

Tree Representations

- Linked
  - **root** variable points to root entry.
  - For each **entry** (starting with **root**),
  - **entry.left** and **entry.right** point to left and right child of **entry**.
  - Either one can be null.
- Array
  - **theElements** is an array of elements.
  - **size** indicate that indices 0 to size-1 are valid.
  - Root is at **theElements[0]**.
  - Left child of **theElements[i]** is at **theElements[$2i+1$]**, but only if $2i+1 <$size. Otherwise, there is no left child.
  - Right child of **theElements[i]** is at **theElements[$2i+2$]**, but only if $2i+2 <$size. Otherwise, there is no right child.
  - For $0 < j <$size, parent of **theElements[j]** is at **theElements[$(j-1)/2$]**.
- Array representation wastes a lot of space unless tree is *complete*.
  - All levels except the bottom level are full.
  - Bottom level is filled from left.

Tree Representations

- Linked
  - **root** variable points to root entry.
  - For each **entry** (starting with **root**),
  - **entry.left** and **entry.right** point to left and right child of **entry**.
  - Either one can be null.
- Array
  - **theElements** is an array of elements.
  - **size** indicate that indices 0 to size-1 are valid.
  - Root is at **theElements[0]**.
  - Left child of **theElements[*i*]** is at **theElements[**$2i+1$**]**, but only if $2i+1<$size. Otherwise, there is no left child.
  - Right child of **theElements[*i*]** is at **theElements[**$2i+2$**]**, but only if $2i+2<$size. Otherwise, there is no right child.
  - For $0<j<$size, parent of **theElements[*j*]** is at **theElements[**$(j-1)/2$**]**.
- Array representation wastes a lot of space unless tree is *complete*.
  - All levels except the bottom level are full.
  - Bottom level is filled from left.
  - Corresponds to using all the elements in 0 to size-1.

Tree Orders

Tree Orders

- Search order.

Tree Orders

- Search order.
  - Every element in left subtree of element is less that that element.

Tree Orders

- Search order.
  - Every element in left subtree of element is less that that element.
  - “Less” means comes alphabetically or numerically earlier.

Tree Orders

- Search order.
  - Every element in left subtree of element is less that that element.
  - “Less” means comes alphabetically or numerically earlier.
  - Every element in right subtree of element is greater that that element.

Tree Orders

- Search order.
  - Every element in left subtree of element is less that that element.
  - “Less” means comes alphabetically or numerically earlier.
  - Every element in right subtree of element is greater that that element.
- Heap order.

Tree Orders

- Search order.
  - Every element in left subtree of element is less that that element.
  - “Less” means comes alphabetically or numerically earlier.
  - Every element in right subtree of element is greater that that element.
- Heap order.
  - Left child is greater or same as element.

Tree Orders

- Search order.
  - Every element in left subtree of element is less that that element.
  - “Less” means comes alphabetically or numerically earlier.
  - Every element in right subtree of element is greater that that element.
- Heap order.
  - Left child is greater or same as element.
  - Right child is greater or same as element.

Applications of orders

Applications of orders

- Search order

Applications of orders

- Search order
  - Good for searching.

Applications of orders

- Search order
  - Good for searching.
  - If you are looking for a key, compare it to root element.

Applications of orders

- Search order
  - Good for searching.
  - If you are looking for a key, compare it to root element.
  - If it is equal, you have found it.

Applications of orders

- Search order
  - Good for searching.
  - If you are looking for a key, compare it to root element.
  - If it is equal, you have found it.
  - If it is less, look in the left subtree.

Applications of orders

- Search order
  - Good for searching.
  - If you are looking for a key, compare it to root element.
  - If it is equal, you have found it.
  - If it is less, look in the left subtree.
  - If it is greater, look in the right subtree.

Applications of orders

- Search order
  - Good for searching.
  - If you are looking for a key, compare it to root element.
  - If it is equal, you have found it.
  - If it is less, look in the left subtree.
  - If it is greater, look in the right subtree.
  - Think: “left is less”.

Applications of orders

- Search order
  - Good for searching.
  - If you are looking for a key, compare it to root element.
  - If it is equal, you have found it.
  - If it is less, look in the left subtree.
  - If it is greater, look in the right subtree.
  - Think: “left is less”.
- Heap order

Applications of orders

- Search order
  - Good for searching.
  - If you are looking for a key, compare it to root element.
  - If it is equal, you have found it.
  - If it is less, look in the left subtree.
  - If it is greater, look in the right subtree.
  - Think: “left is less”.
- Heap order
  - Root is least element.

Applications of orders

- Search order
  - Good for searching.
  - If you are looking for a key, compare it to root element.
  - If it is equal, you have found it.
  - If it is less, look in the left subtree.
  - If it is greater, look in the right subtree.
  - Think: "left is less".
- Heap order
  - Root is least element.
  - It's at the "top of the heap".

Applications of orders

- Search order
  - Good for searching.
  - If you are looking for a key, compare it to root element.
  - If it is equal, you have found it.
  - If it is less, look in the left subtree.
  - If it is greater, look in the right subtree.
  - Think: “left is less”.
- Heap order
  - Root is least element.
  - It’s at the “top of the heap”.
  - Good for *priority queue*.

Applications of orders

- Search order
  - Good for searching.
  - If you are looking for a key, compare it to root element.
  - If it is equal, you have found it.
  - If it is less, look in the left subtree.
  - If it is greater, look in the right subtree.
  - Think: “left is less”.
- Heap order
  - Root is least element.
  - It’s at the “top of the heap”.
  - Good for *priority queue*.
  - Serve elements in order of key.

Applications of orders

- Search order
  - Good for searching.
  - If you are looking for a key, compare it to root element.
  - If it is equal, you have found it.
  - If it is less, look in the left subtree.
  - If it is greater, look in the right subtree.
  - Think: "left is less".
- Heap order
  - Root is least element.
  - It's at the "top of the heap".
  - Good for *priority queue*.
  - Serve elements in order of key.
  - For example, at a hospital emergency room serve in order of minutes until death.

More notes

More notes

- Blackboard doesn't use a fixed width font.

More notes

- Blackboard doesn't use a fixed width font.
- So I will put the notes into text files.

More notes

- Blackboard doesn't use a fixed width font.
- So I will put the notes into text files.
- Red font means hyperlink.

More notes

- Blackboard doesn't use a fixed width font.
- So I will put the notes into text files.
- Red font means hyperlink.
  - Notes on binary search trees.

More notes

- Blackboard doesn't use a fixed width font.
- So I will put the notes into text files.
- Red font means hyperlink.
  - Notes on binary search trees.
  - Notes on heaps.

Running Times

Suppose we are using a binary tree to implement a Map and the entries are inserted in random order.

Running Times

Suppose we are using a binary tree to implement a Map and the entries are inserted in random order.
How many comparisons does it take to find an entry?

Running Times

Suppose we are using a binary tree to implement a Map and the entries are inserted in random order.
How many comparisons does it take to find an entry?
Find key in tree with n entries:

Running Times

Suppose we are using a binary tree to implement a Map and the entries are inserted in random order.
How many comparisons does it take to find an entry?
Find key in tree with n entries:

- Compare key with root.key.

Running Times

Suppose we are using a binary tree to implement a Map and the entries are inserted in random order.
How many comparisons does it take to find an entry?
Find key in tree with n entries:

- Compare key with root.key.
- Find key in left or right tree with up to n/2 entries (on average):

Running Times

Suppose we are using a binary tree to implement a Map and the entries are inserted in random order.
How many comparisons does it take to find an entry?
Find key in tree with n entries:

- Compare key with root.key.
- Find key in left or right tree with up to n/2 entries (on average):
  - Compare key with root.key.

Running Times

Suppose we are using a binary tree to implement a Map and the entries are inserted in random order.
How many comparisons does it take to find an entry?
Find key in tree with n entries:

- Compare key with root.key.
- Find key in left or right tree with up to n/2 entries (on average):
  - Compare key with root.key.
  - Find key in left or right tree with up to n/4 entries (on average):

Running Times

Suppose we are using a binary tree to implement a Map and the entries are inserted in random order.
How many comparisons does it take to find an entry?
Find key in tree with n entries:

- Compare key with root.key.
- Find key in left or right tree with up to n/2 entries (on average):
  - Compare key with root.key.
  - Find key in left or right tree with up to n/4 entries (on average):
    - Compare key with root.key.

Running Times

Suppose we are using a binary tree to implement a Map and the entries are inserted in random order.
How many comparisons does it take to find an entry?
Find key in tree with n entries:

- Compare key with root.key.
- Find key in left or right tree with up to n/2 entries (on average):
  - Compare key with root.key.
  - Find key in left or right tree with up to n/4 entries (on average):
    - Compare key with root.key.
    - Find key in left or right tree with up to n/8 entries (on average):

Running Times

Suppose we are using a binary tree to implement a Map and the entries are inserted in random order.
How many comparisons does it take to find an entry?
Find key in tree with n entries:

- Compare key with root.key.
- Find key in left or right tree with up to n/2 entries (on average):
  - Compare key with root.key.
  - Find key in left or right tree with up to n/4 entries (on average):
    - Compare key with root.key.
    - Find key in left or right tree with up to n/8 entries (on average):

This is our gold coin example again.

Running Times

Suppose we are using a binary tree to implement a Map and the entries are inserted in random order.
How many comparisons does it take to find an entry?
Find key in tree with n entries:

- Compare key with root.key.
- Find key in left or right tree with up to n/2 entries (on average):
  - Compare key with root.key.
  - Find key in left or right tree with up to n/4 entries (on average):
    - Compare key with root.key.
    - Find key in left or right tree with up to n/8 entries (on average):

This is our gold coin example again.

- We know there will be about $\log_2 n$ comparisons.

Running Times

Suppose we are using a binary tree to implement a Map and the entries are inserted in random order.
How many comparisons does it take to find an entry?
Find key in tree with n entries:

- Compare key with root.key.
- Find key in left or right tree with up to n/2 entries (on average):
  - Compare key with root.key.
  - Find key in left or right tree with up to n/4 entries (on average):
    - Compare key with root.key.
    - Find key in left or right tree with up to n/8 entries (on average):

This is our gold coin example again.

- We know there will be about $\log_2 n$ comparisons.
- So the running time is $\mathrm{O}(\log n)$.

Running Times

Suppose we are using a binary tree to implement a Map and the entries are inserted in random order.
How many comparisons does it take to find an entry?
Find key in tree with n entries:

- Compare key with root.key.
- Find key in left or right tree with up to n/2 entries (on average):
  - Compare key with root.key.
  - Find key in left or right tree with up to n/4 entries (on average):
    - Compare key with root.key.
    - Find key in left or right tree with up to n/8 entries (on average):

This is our gold coin example again.

- We know there will be about $\log_2 n$ comparisons.
- So the running time is $\mathrm{O}(\log n)$.
- ON AVERAGE. IF INPUTS ARE RANDOMLY ORDERED.

Worst case for search trees

Worst case for search trees

The situation is bad if inputs are already ordered by key.

Worst case for search trees

The situation is bad if inputs are already ordered by key.
In that case, everything is always on the right:

Worst case for search trees

The situation is bad if inputs are already ordered by key.
In that case, everything is always on the right:
Find key in tree with n entries:

Worst case for search trees

The situation is bad if inputs are already ordered by key.
In that case, everything is always on the right:
Find key in tree with n entries:

- Compare key with root.key.

Worst case for search trees

The situation is bad if inputs are already ordered by key.
In that case, everything is always on the right:
Find key in tree with n entries:

- Compare key with root.key.
- Find key in right tree with up to n-1 entries:

Worst case for search trees

The situation is bad if inputs are already ordered by key.
In that case, everything is always on the right:
Find key in tree with n entries:

- Compare key with root.key.
- Find key in right tree with up to n-1 entries:
  - Compare key with root.key.

Worst case for search trees

The situation is bad if inputs are already ordered by key.
In that case, everything is always on the right:
Find key in tree with n entries:

- Compare key with root.key.
- Find key in right tree with up to n-1 entries:
  - Compare key with root.key.
  - Find key in right tree with up to n-2 entries:

Worst case for search trees

The situation is bad if inputs are already ordered by key.
In that case, everything is always on the right:
Find key in tree with n entries:

- Compare key with root.key.
- Find key in right tree with up to n-1 entries:
  - Compare key with root.key.
  - Find key in right tree with up to n-2 entries:
    - Compare key with root.key.

Worst case for search trees

The situation is bad if inputs are already ordered by key.
In that case, everything is always on the right:
Find key in tree with n entries:

- Compare key with root.key.
- Find key in right tree with up to n-1 entries:
  - Compare key with root.key.
  - Find key in right tree with up to n-2 entries:
    - Compare key with root.key.
    - Find key in right tree with up to n-3 entries:

Worst case for search trees

The situation is bad if inputs are already ordered by key.
In that case, everything is always on the right:
Find key in tree with n entries:

- Compare key with root.key.
- Find key in right tree with up to n-1 entries:
  - Compare key with root.key.
  - Find key in right tree with up to n-2 entries:
    - Compare key with root.key.
    - Find key in right tree with up to n-3 entries:

This is going to require n comparisons for a running time of $\mathrm{O}(n)$.

Worst case for search trees

The situation is bad if inputs are already ordered by key.
In that case, everything is always on the right:
Find key in tree with n entries:

- Compare key with root.key.
- Find key in right tree with up to n-1 entries:
  - Compare key with root.key.
  - Find key in right tree with up to n-2 entries:
    - Compare key with root.key.
    - Find key in right tree with up to n-3 entries:

This is going to require n comparisons for a running time of $\mathrm{O}(n)$.
Can this happen?

Worst case for search trees

The situation is bad if inputs are already ordered by key.
In that case, everything is always on the right:
Find key in tree with n entries:

- Compare key with root.key.
- Find key in right tree with up to n-1 entries:
  - Compare key with root.key.
  - Find key in right tree with up to n-2 entries:
    - Compare key with root.key.
    - Find key in right tree with up to n-3 entries:

This is going to require n comparisons for a running time of $\mathrm{O}(n)$.
Can this happen?

- Yes, someone might order the inputs by key in an effort to be helpful.

Worst case for search trees

The situation is bad if inputs are already ordered by key.
In that case, everything is always on the right:
Find key in tree with n entries:

- Compare key with root.key.
- Find key in right tree with up to n-1 entries:
  - Compare key with root.key.
  - Find key in right tree with up to n-2 entries:
    - Compare key with root.key.
    - Find key in right tree with up to n-3 entries:

This is going to require n comparisons for a running time of $\mathrm{O}(n)$.
Can this happen?

- Yes, someone might order the inputs by key in an effort to be helpful.
- If your English professor asks for an example for irony, you can offer this one!

Worst case for search trees

The situation is bad if inputs are already ordered by key.
In that case, everything is always on the right:
Find key in tree with n entries:

- Compare key with root.key.
- Find key in right tree with up to n-1 entries:
  - Compare key with root.key.
  - Find key in right tree with up to n-2 entries:
    - Compare key with root.key.
    - Find key in right tree with up to n-3 entries:

This is going to require n comparisons for a running time of $\mathrm{O}(n)$.
Can this happen?

- Yes, someone might order the inputs by key in an effort to be helpful.
- If your English professor asks for an example for irony, you can offer this one!
- For example, what if you read in a phone directory that is already sorted by name?

Worst case for search trees

The situation is bad if inputs are already ordered by key.
In that case, everything is always on the right:
Find key in tree with n entries:

- Compare key with root.key.
- Find key in right tree with up to n-1 entries:
  - Compare key with root.key.
  - Find key in right tree with up to n-2 entries:
    - Compare key with root.key.
    - Find key in right tree with up to n-3 entries:

This is going to require n comparisons for a running time of $\mathrm{O}(n)$.
Can this happen?

- Yes, someone might order the inputs by key in an effort to be helpful.
- If your English professor asks for an example for irony, you can offer this one!
- For example, what if you read in a phone directory that is already sorted by name?

Can it be fixed?

Worst case for search trees

The situation is bad if inputs are already ordered by key.
In that case, everything is always on the right:
Find key in tree with n entries:

- Compare key with root.key.
- Find key in right tree with up to n-1 entries:
  - Compare key with root.key.
  - Find key in right tree with up to n-2 entries:
    - Compare key with root.key.
    - Find key in right tree with up to n-3 entries:

This is going to require *n* comparisons for a running time of $\mathrm{O}(n)$.
Can this happen?

- Yes, someone might order the inputs by key in an effort to be helpful.
- If your English professor asks for an example for irony, you can offer this one!
- For example, what if you read in a phone directory that is already sorted by name?

Can it be fixed?

- Yes, using a *balanced* tree.

Worst case for search trees

The situation is bad if inputs are already ordered by key.
In that case, everything is always on the right:
Find key in tree with n entries:

- Compare key with root.key.
- Find key in right tree with up to n-1 entries:
  - Compare key with root.key.
  - Find key in right tree with up to n-2 entries:
    - Compare key with root.key.
    - Find key in right tree with up to n-3 entries:

This is going to require n comparisons for a running time of $O(n)$.
Can this happen?

- Yes, someone might order the inputs by key in an effort to be helpful.
- If your English professor asks for an example for irony, you can offer this one!
- For example, what if you read in a phone directory that is already sorted by name?

Can it be fixed?

- Yes, using a *balanced* tree.
- It's a bit complicated and will have to wait until CSC317.

Worst case for search trees

The situation is bad if inputs are already ordered by key.
In that case, everything is always on the right:
Find key in tree with n entries:

- Compare key with root.key.
- Find key in right tree with up to n-1 entries:
  - Compare key with root.key.
  - Find key in right tree with up to n-2 entries:
    - Compare key with root.key.
    - Find key in right tree with up to n-3 entries:

This is going to require n comparisons for a running time of $O(n)$.
Can this happen?

- Yes, someone might order the inputs by key in an effort to be helpful.
- If your English professor asks for an example for irony, you can offer this one!
- For example, what if you read in a phone directory that is already sorted by name?

Can it be fixed?

- Yes, using a *balanced* tree.
- It's a bit complicated and will have to wait until CSC317.
- For java.util.TreeMap, get, put, and remove are all $O(\log n)$.

Running Time for Heap

Running Time for Heap

- The situation with heaps is much better.

Running Time for Heap

- The situation with heaps is much better.
- Neither randomness nor CSC317 is required.

Running Time for Heap

- The situation with heaps is much better.
- Neither randomness nor CSC317 is required.
- Let's figure out how many levels there are in the tree for a given number of entries:

Running Time for Heap

- The situation with heaps is much better.
- Neither randomness nor CSC317 is required.
- Let's figure out how many levels there are in the tree for a given number of entries:
  - 1 entry means 1 level.

Running Time for Heap

- The situation with heaps is much better.
- Neither randomness nor CSC317 is required.
- Let's figure out how many levels there are in the tree for a given number of entries:
  - 1 entry means 1 level.
  - 2-3 entries means 2 levels.

Running Time for Heap

- The situation with heaps is much better.
- Neither randomness nor CSC317 is required.
- Let's figure out how many levels there are in the tree for a given number of entries:
  - 1 entry means 1 level.
  - 2-3 entries means 2 levels.
  - 4-7 entries means 3 levels.

Running Time for Heap

- The situation with heaps is much better.
- Neither randomness nor CSC317 is required.
- Let's figure out how many levels there are in the tree for a given number of entries:
  - 1 entry means 1 level.
  - 2-3 entries means 2 levels.
  - 4-7 entries means 3 levels.
  - 8-15 entries means 4 levels.

Running Time for Heap

- The situation with heaps is much better.
- Neither randomness nor CSC317 is required.
- Let's figure out how many levels there are in the tree for a given number of entries:
  - 1 entry means 1 level.
  - 2-3 entries means 2 levels.
  - 4-7 entries means 3 levels.
  - 8-15 entries means 4 levels.
  - 16-31 entries means 5 levels.

Running Time for Heap

- The situation with heaps is much better.
- Neither randomness nor CSC317 is required.
- Let's figure out how many levels there are in the tree for a given number of entries:
  - 1 entry means 1 level.
  - 2-3 entries means 2 levels.
  - 4-7 entries means 3 levels.
  - 8-15 entries means 4 levels.
  - 16-31 entries means 5 levels.
  - n entries means up to $\log_2 n + 1$ levels.

Running Time for Heap

- The situation with heaps is much better.
- Neither randomness nor CSC317 is required.
- Let's figure out how many levels there are in the tree for a given number of entries:
  - 1 entry means 1 level.
  - 2-3 entries means 2 levels.
  - 4-7 entries means 3 levels.
  - 8-15 entries means 4 levels.
  - 16-31 entries means 5 levels.
  - n entries means up to $\log_2 n + 1$ levels.
- Cost of **offer**

Running Time for Heap

- The situation with heaps is much better.
- Neither randomness nor CSC317 is required.
- Let's figure out how many levels there are in the tree for a given number of entries:
  - 1 entry means 1 level.
  - 2-3 entries means 2 levels.
  - 4-7 entries means 3 levels.
  - 8-15 entries means 4 levels.
  - 16-31 entries means 5 levels.
  - n entries means up to $\log_2 n + 1$ levels.
- Cost of **offer**
  - Start at bottom.

Running Time for Heap

- The situation with heaps is much better.
- Neither randomness nor CSC317 is required.
- Let's figure out how many levels there are in the tree for a given number of entries:
  - 1 entry means 1 level.
  - 2-3 entries means 2 levels.
  - 4-7 entries means 3 levels.
  - 8-15 entries means 4 levels.
  - 16-31 entries means 5 levels.
  - n entries means up to $\log_2 n + 1$ levels.
- Cost of **offer**
  - Start at bottom.
  - Perhaps swap with the parent all the way to the top.

Running Time for Heap

- The situation with heaps is much better.
- Neither randomness nor CSC317 is required.
- Let's figure out how many levels there are in the tree for a given number of entries:
  - 1 entry means 1 level.
  - 2-3 entries means 2 levels.
  - 4-7 entries means 3 levels.
  - 8-15 entries means 4 levels.
  - 16-31 entries means 5 levels.
  - n entries means up to $\log_2 n + 1$ levels.
- Cost of **offer**
  - Start at bottom.
  - Perhaps swap with the parent all the way to the top.
  - But that is levels-1 swaps at most,

Running Time for Heap

- The situation with heaps is much better.
- Neither randomness nor CSC317 is required.
- Let's figure out how many levels there are in the tree for a given number of entries:
  - 1 entry means 1 level.
  - 2-3 entries means 2 levels.
  - 4-7 entries means 3 levels.
  - 8-15 entries means 4 levels.
  - 16-31 entries means 5 levels.
  - n entries means up to $\log_2 n + 1$ levels.
- Cost of **offer**
  - Start at bottom.
  - Perhaps swap with the parent all the way to the top.
  - But that is levels-1 swaps at most,
  - so at most $\log_2 n$ swaps.

Running Time for Heap

- The situation with heaps is much better.
- Neither randomness nor CSC317 is required.
- Let's figure out how many levels there are in the tree for a given number of entries:
  - 1 entry means 1 level.
  - 2-3 entries means 2 levels.
  - 4-7 entries means 3 levels.
  - 8-15 entries means 4 levels.
  - 16-31 entries means 5 levels.
  - n entries means up to $\log_2 n + 1$ levels.
- Cost of **offer**
  - Start at bottom.
  - Perhaps swap with the parent all the way to the top.
  - But that is levels-1 swaps at most,
  - so at most $\log_2 n$ swaps.
  - The running time is $\mathrm{O}(\log n)$.

Running Time for Heap

- The situation with heaps is much better.
- Neither randomness nor CSC317 is required.
- Let's figure out how many levels there are in the tree for a given number of entries:
  - 1 entry means 1 level.
  - 2-3 entries means 2 levels.
  - 4-7 entries means 3 levels.
  - 8-15 entries means 4 levels.
  - 16-31 entries means 5 levels.
  - n entries means up to $\log_2 n + 1$ levels.
- Cost of **offer**
  - Start at bottom.
  - Perhaps swap with the parent all the way to the top.
  - But that is levels-1 swaps at most,
  - so at most $\log_2 n$ swaps.
  - The running time is $\mathrm{O}(\log n)$.
- Cost of **poll**.

Running Time for Heap

- The situation with heaps is much better.
- Neither randomness nor CSC317 is required.
- Let's figure out how many levels there are in the tree for a given number of entries:
  - 1 entry means 1 level.
  - 2-3 entries means 2 levels.
  - 4-7 entries means 3 levels.
  - 8-15 entries means 4 levels.
  - 16-31 entries means 5 levels.
  - n entries means up to $\log_2 n + 1$ levels.
- Cost of **offer**
  - Start at bottom.
  - Perhaps swap with the parent all the way to the top.
  - But that is levels-1 swaps at most,
  - so at most $\log_2 n$ swaps.
  - The running time is $\mathrm{O}(\log n)$.
- Cost of **poll**.
  - Swap with a child up to $\log_2 n$ times

Running Time for Heap

- The situation with heaps is much better.
- Neither randomness nor CSC317 is required.
- Let's figure out how many levels there are in the tree for a given number of entries:
  - 1 entry means 1 level.
  - 2-3 entries means 2 levels.
  - 4-7 entries means 3 levels.
  - 8-15 entries means 4 levels.
  - 16-31 entries means 5 levels.
  - n entries means up to $\log_2 n + 1$ levels.
- Cost of **offer**
  - Start at bottom.
  - Perhaps swap with the parent all the way to the top.
  - But that is levels-1 swaps at most,
  - so at most $\log_2 n$ swaps.
  - The running time is $\mathrm{O}(\log n)$.
- Cost of **poll**.
  - Swap with a child up to $\log_2 n$ times
  - The running time is $\mathrm{O}(\log n)$.

Running Time for Heap

- The situation with heaps is much better.
- Neither randomness nor CSC317 is required.
- Let's figure out how many levels there are in the tree for a given number of entries:
  - 1 entry means 1 level.
  - 2-3 entries means 2 levels.
  - 4-7 entries means 3 levels.
  - 8-15 entries means 4 levels.
  - 16-31 entries means 5 levels.
  - n entries means up to $\log_2 n + 1$ levels.
- Cost of **offer**
  - Start at bottom.
  - Perhaps swap with the parent all the way to the top.
  - But that is levels-1 swaps at most,
  - so at most $\log_2 n$ swaps.
  - The running time is $\mathrm{O}(\log n)$.
- Cost of **poll**.
  - Swap with a child up to $\log_2 n$ times
  - The running time is $\mathrm{O}(\log n)$.
- **peek** is obviously $\mathrm{O}(1)$.

Summary

Summary

- Tree is like list but with up to two successors per element called *left* and *right*.

Summary

- Tree is like list but with up to two successors per element called *left* and *right*.
- New terms: *root*, *leaf*, *child*, *parent*, *subtree*, *depth*, *height*.

Summary

- Tree is like list but with up to two successors per element called *left* and *right*.
- New terms: *root*, *leaf*, *child*, *parent*, *subtree*, *depth*, *height*.
- Linked representation is pretty obvious.

Summary

- Tree is like list but with up to two successors per element called *left* and *right*.
- New terms: *root*, *leaf*, *child*, *parent*, *subtree*, *depth*, *height*.
- Linked representation is pretty obvious.
- Array representation is a bit more tricky.

Summary

- Tree is like list but with up to two successors per element called *left* and *right*.
- New terms: *root*, *leaf*, *child*, *parent*, *subtree*, *depth*, *height*.
- Linked representation is pretty obvious.
- Array representation is a bit more tricky.
- Array representation only makes sense for *complete* trees.

Summary

- Tree is like list but with up to two successors per element called *left* and *right*.
- New terms: *root*, *leaf*, *child*, *parent*, *subtree*, *depth*, *height*.
- Linked representation is pretty obvious.
- Array representation is a bit more tricky.
- Array representation only makes sense for *complete* trees.
- Trees can have *search order* or *heap order*.

Summary

- Tree is like list but with up to two successors per element called *left* and *right*.
- New terms: *root*, *leaf*, *child*, *parent*, *subtree*, *depth*, *height*.
- Linked representation is pretty obvious.
- Array representation is a bit more tricky.
- Array representation only makes sense for *complete* trees.
- Trees can have *search order* or *heap order*.
- Search order good for searching.

Summary

- Tree is like list but with up to two successors per element called *left* and *right*.
- New terms: *root*, *leaf*, *child*, *parent*, *subtree*, *depth*, *height*.
- Linked representation is pretty obvious.
- Array representation is a bit more tricky.
- Array representation only makes sense for *complete* trees.
- Trees can have *search order* or *heap order*.
- Search order good for searching.
  - Find, add, remove all $O(\log n)$ if elements inserted in random order.

Summary

- Tree is like list but with up to two successors per element called *left* and *right*.
- New terms: *root*, *leaf*, *child*, *parent*, *subtree*, *depth*, *height*.
- Linked representation is pretty obvious.
- Array representation is a bit more tricky.
- Array representation only makes sense for *complete* trees.
- Trees can have *search order* or *heap order*.
- Search order good for searching.
  - Find, add, remove all $O(\log n)$ if elements inserted in random order.
  - Find, add, remove all $O(n)$ if elements inserted in sorted order.

Summary

- Tree is like list but with up to two successors per element called *left* and *right*.
- New terms: *root*, *leaf*, *child*, *parent*, *subtree*, *depth*, *height*.
- Linked representation is pretty obvious.
- Array representation is a bit more tricky.
- Array representation only makes sense for *complete* trees.
- Trees can have *search order* or *heap order*.
- Search order good for searching.
  - Find, add, remove all $\mathrm{O}(\log n)$ if elements inserted in random order.
  - Find, add, remove all $\mathrm{O}(n)$ if elements inserted in sorted order.
- Heap order good for priority queue.

Summary

- Tree is like list but with up to two successors per element called *left* and *right*.
- New terms: *root*, *leaf*, *child*, *parent*, *subtree*, *depth*, *height*.
- Linked representation is pretty obvious.
- Array representation is a bit more tricky.
- Array representation only makes sense for *complete* trees.
- Trees can have *search order* or *heap order*.
- Search order good for searching.
  - Find, add, remove all $\mathrm{O}(\log n)$ if elements inserted in random order.
  - Find, add, remove all $\mathrm{O}(n)$ if elements inserted in sorted order.
- Heap order good for priority queue.
  - Peek is $\mathrm{O}(1)$.

Summary

- Tree is like list but with up to two successors per element called *left* and *right*.
- New terms: *root*, *leaf*, *child*, *parent*, *subtree*, *depth*, *height*.
- Linked representation is pretty obvious.
- Array representation is a bit more tricky.
- Array representation only makes sense for *complete* trees.
- Trees can have *search order* or *heap order*.
- Search order good for searching.
  - Find, add, remove all $\mathrm{O}(\log n)$ if elements inserted in random order.
  - Find, add, remove all $\mathrm{O}(n)$ if elements inserted in sorted order.
- Heap order good for priority queue.
  - Peek is $\mathrm{O}(1)$.
  - Offer and poll are $\mathrm{O}(\log n)$.