Balanced Trees

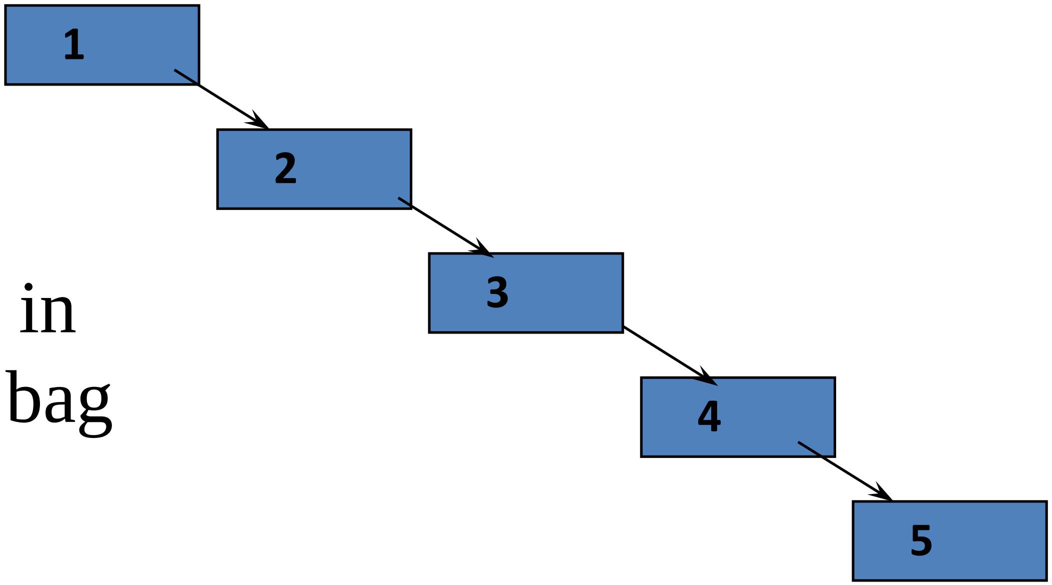
Topics

- Why B-Tree
- The B-Tree Rules
- The Set Class ADT with B-Trees
- Search for an Item in a B-Tree
- Insert an Item in a B-Tree (*)
- Remove a Item from a B-Tree (*)

B-TREES AND THE SET CLASS

The problem of an unbalanced BST

- Maximum depth of a BST with n entires: n-1



Insert 1, 2, 3,4,5 in that order into a bag using a BST

Worst-Case Times for BSTs

- Adding, deleting or searching for an entry in a BST with n entries is $O(d)$ in the worst case, where d is the depth of the BST
- Since d is no more than n-1, the operations in the worst case is $O(n-1)$.
- Conclusion: the worst case time for the add, delete or search operation of a BST is $O(n)$

The B-Tree Basics

- Similar to a binary search tree (BST)
  - where the implementation requires the ability to compare two entries via a ***less-than operator*** **(<)**
- But a B-tree is NOT a BST – in fact it is not even a binary tree
  - *B-tree nodes have many (more than two) children*
  - *each node contains more than just a single entry*
- Advantages:
  - *Easy to search, and not too deep*

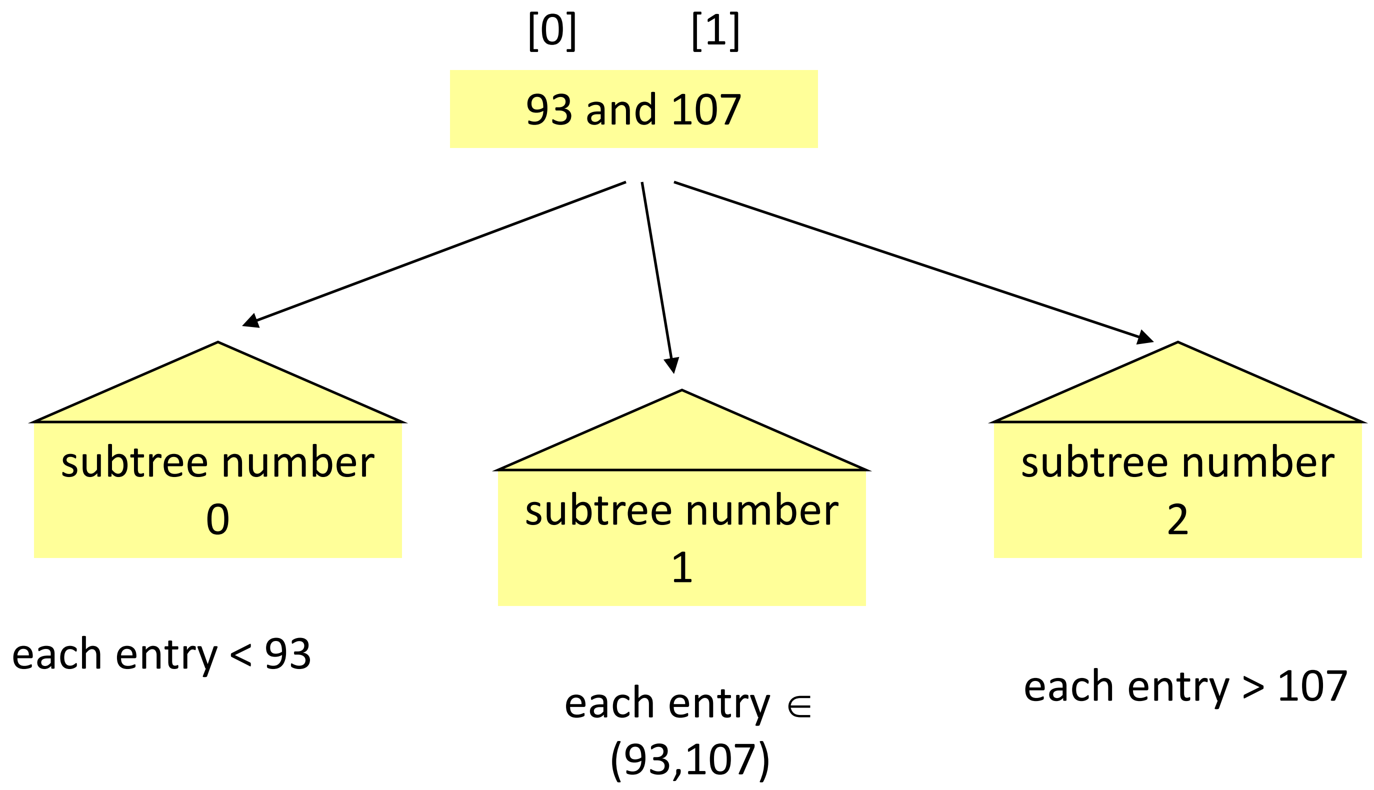
The B-Tree Rules

- The entries in a B-tree node
  - B-tree Rule 1: The root may have as few as one entry (or 0 entry if no children); every other node has at least MINIMUM entries
  - B-tree Rule 2: The maximum number of entries in a node is 2* MINIMUM.
  - B-tree Rule 3: The entries of each B-tree node are stored in a partially filled array, sorted from the smallest to the largest.

The B-Tree Rules (cont.)

- The subtrees below a B-tree node
  - B-tree Rule 4: The number of the subtrees below a non-leaf node with n entries is always $n+1$
  - B-tree Rule 5: For any non-leaf node:
    - ✓ (a) An entry at index i is greater than all the entries in subtree number i of the node
    - ✓ (b) An entry at index i is less than all the entries in subtree number $i+1$ of the node

An Example of B-Tree

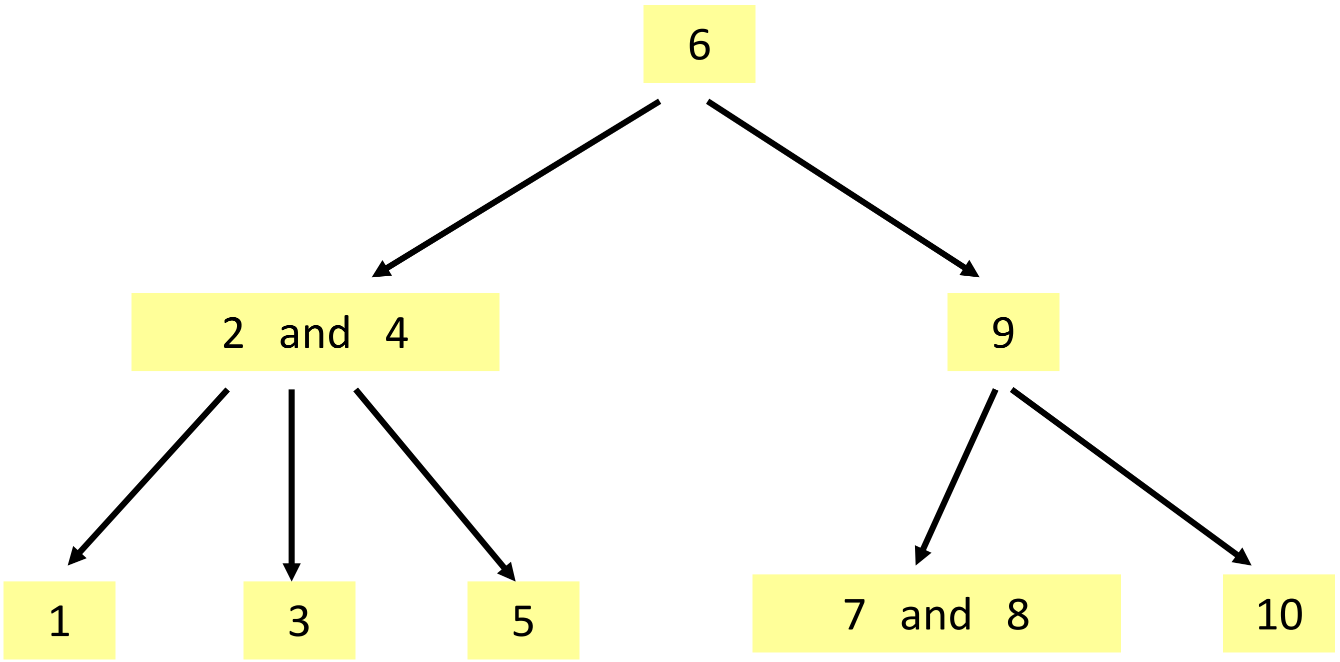


The B-Tree Rules (cont.)

- A B-tree is balanced
  - B-tree Rule 6: Every leaf in a B-tree has the same depth

- This rule ensures that a B-tree is balanced

Another Example, MINIMUM = 1



Can you verify that all 6 rules are satisfied?

The set ADT with a B-Tree

- Combine fixed size array with linked nodes
  - data[]
  - *subset[]
- number of entries vary
  - data_count
- number of children vary
  - child_count
  - = data_count+1?

```
template <class Item>
class set
{
public:
            ... ...
    bool insert(const Item& entry);
    std::size_t erase(const Item& target);
    std::size_t count(const Item& target) const;
private:
    // MEMBER CONSTANTS
    static const std::size_t MINIMUM = 200;
    static const std::size_t MAXIMUM = 2 * MINIMUM;
    // MEMBER VARIABLES
    std::size_t data_count;
    Item data[MAXIMUM+1]; // why +1? -for insert/erase
    std::size_t child_count;
    set *subset[MAXIMUM+2]; // why +2? - one more

};
```

Invariant for the set Class

- The entries of a set is stored in a B-tree, satisfying the six B-tree rules.
- The number of entries in a node is stored in data_count, and the entries are stored in data[0] through data[data_count-1]
- The number of subtrees of a node is stored in child_count, and the subtrees are pointed by set pointers subset[0] through subset[child_count-1]

Search for an Item in a B-Tree

- Prototype:
  - std::size_t count(const Item& target) const;

- Post-condition:
  - Returns the number of items equal to the target
  - (either 0 or 1 for a set).

Searching for an Item: count

search for 10: cout << count (10);

Start at the root.

1) locate i so
   that !(data[i]<target)
2) If (data[i] is target)

   return 1;

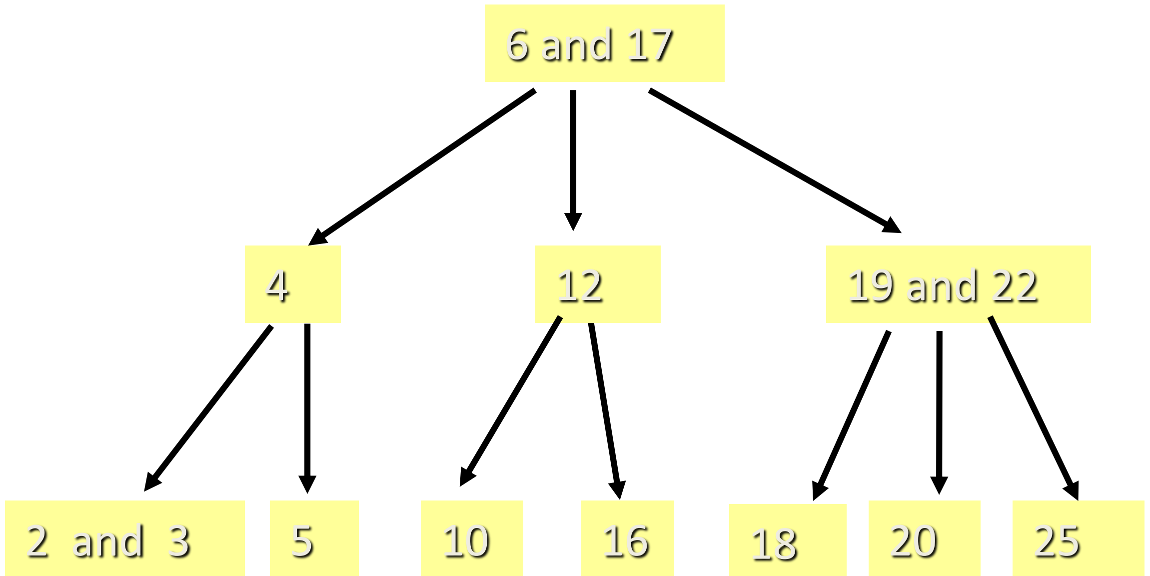
   else if (no children)

   return 0;

   else

   return

   subset[i]->count (target);



Searching for an Item: count

search for 10:   cout << count (10);

Start at the root.

1) locate i so that !(data[i]<target)
2) If (data[i] is target)

        return 1;

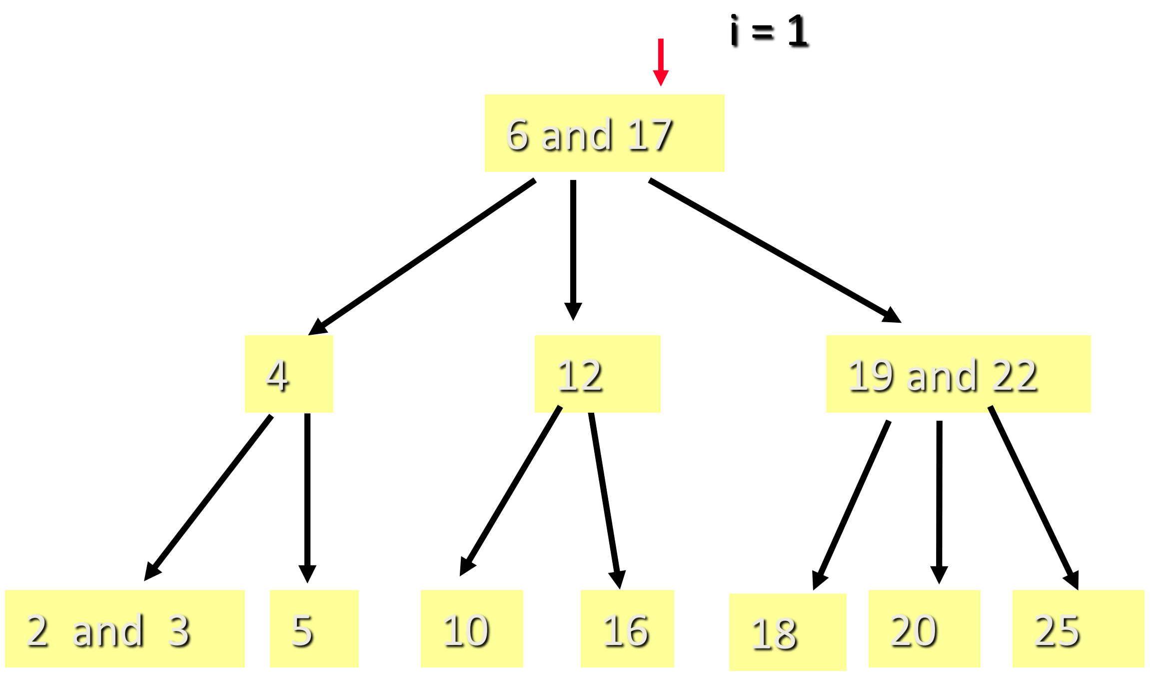
    else if (no children)

        return 0;

    else

        return

        subset[i]->count (target);



Searching for an Item: count

search for 10: cout << count (10);

Start at the root.

1) locate i so that !(data[i]<target)
2) If (data[i] is target)

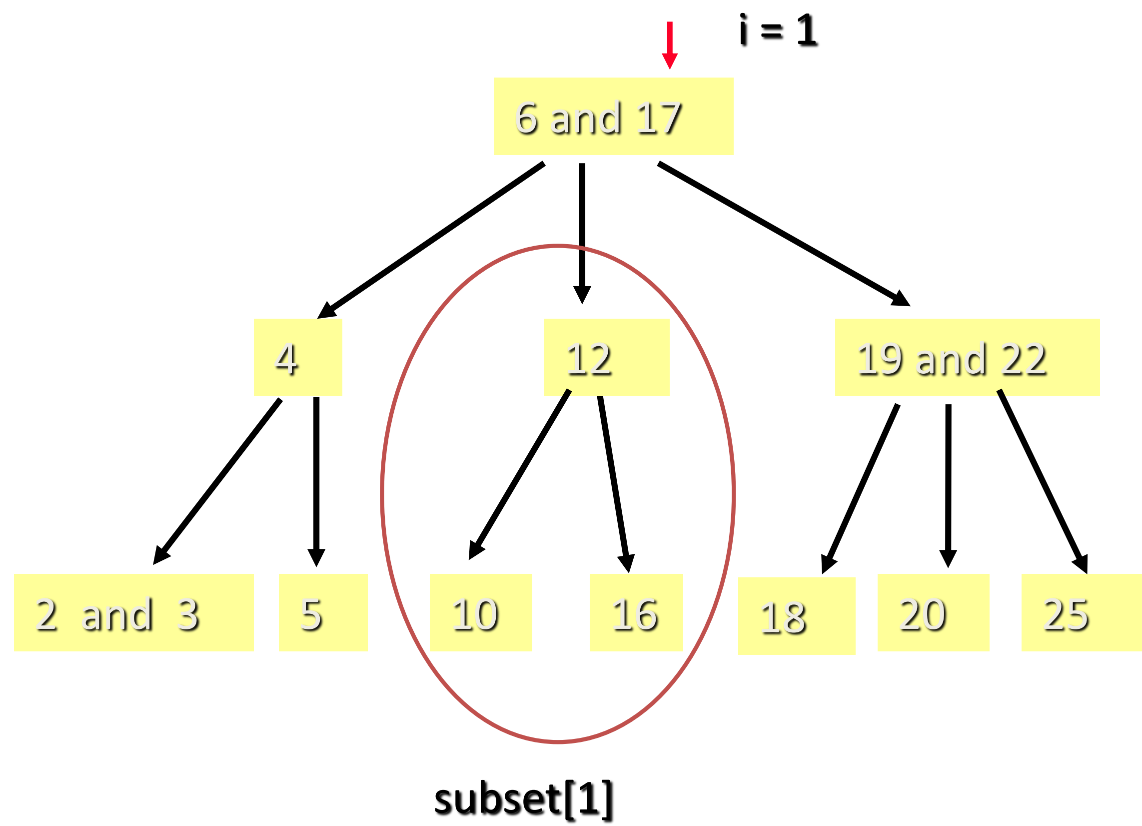
        return 1;

    else if (no children)

        return 0;

    else

        return subset[i]->count (target);



Searching for an Item: count

search for 10: cout << count (10);

Start at the root.

1) locate i so that !(data[i]<target)

2) If (data[i] is target)

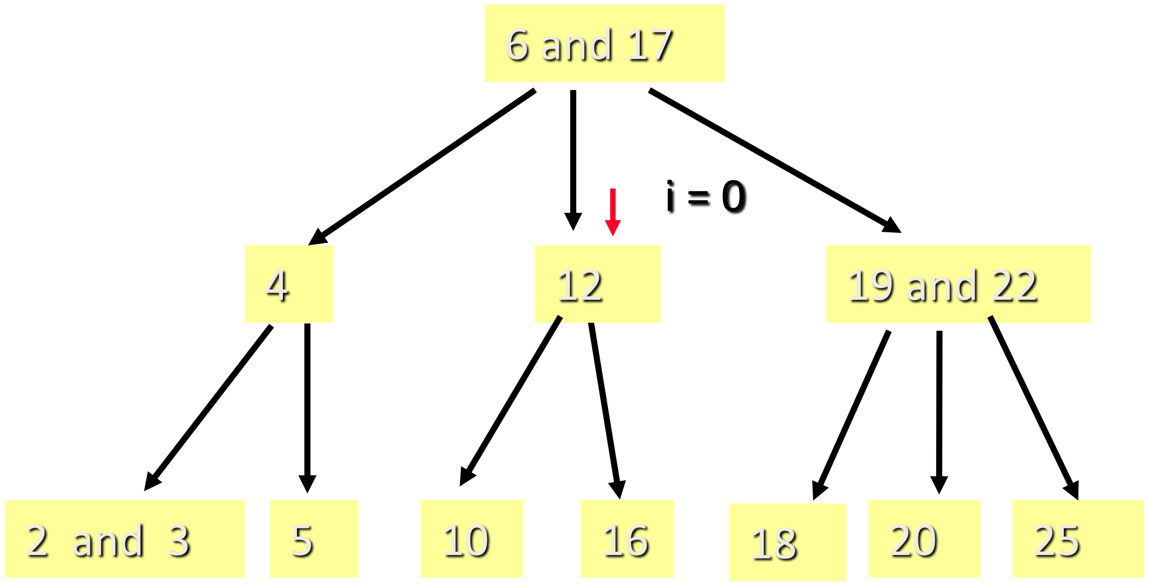
    return 1;

  else if (no children)

    return 0;

  else

    return subset[i]->count (target);



Searching for an Item: count

search for 10: cout << count (10);

Start at the root.

1) locate i so that !(data[i]<target)

2) If (data[i] is target)

        return 1;

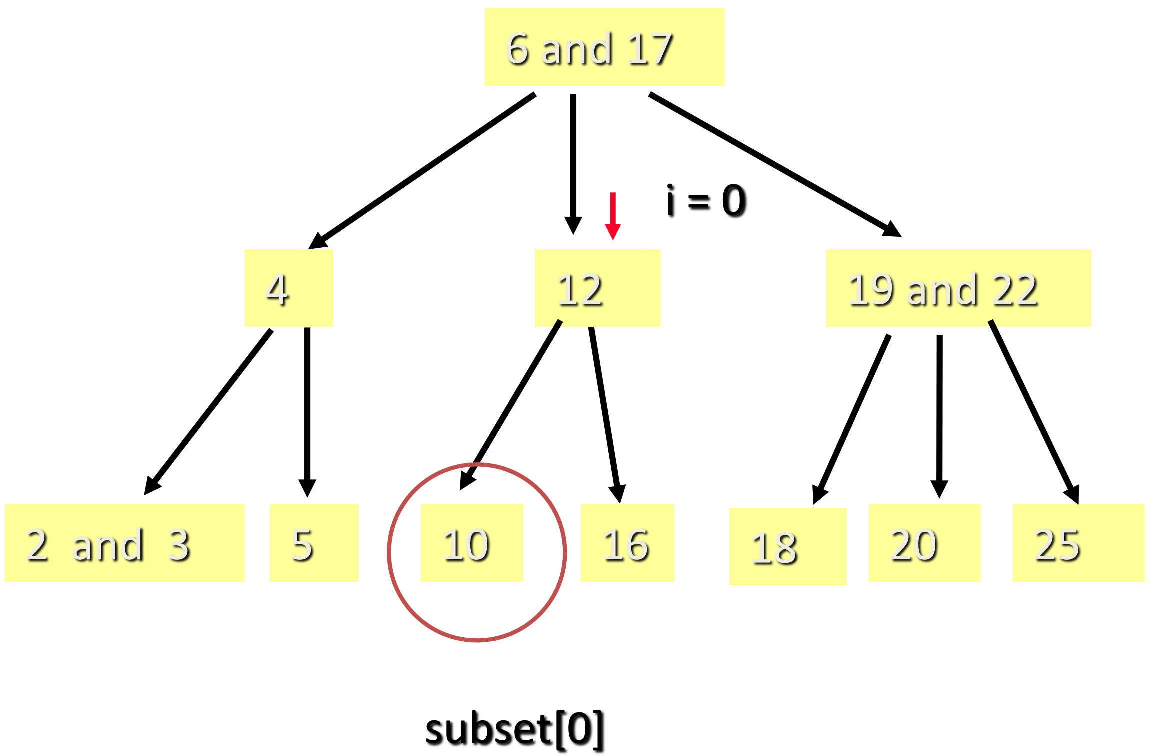
    else if (no children)

        return 0;

    else

        return

        subset[i]->count (target);



Searching for an Item: count

search for 10: cout << count (10);

Start at the root.

1) locate i so that !(data[i]<target)
2) If (data[i] is target)

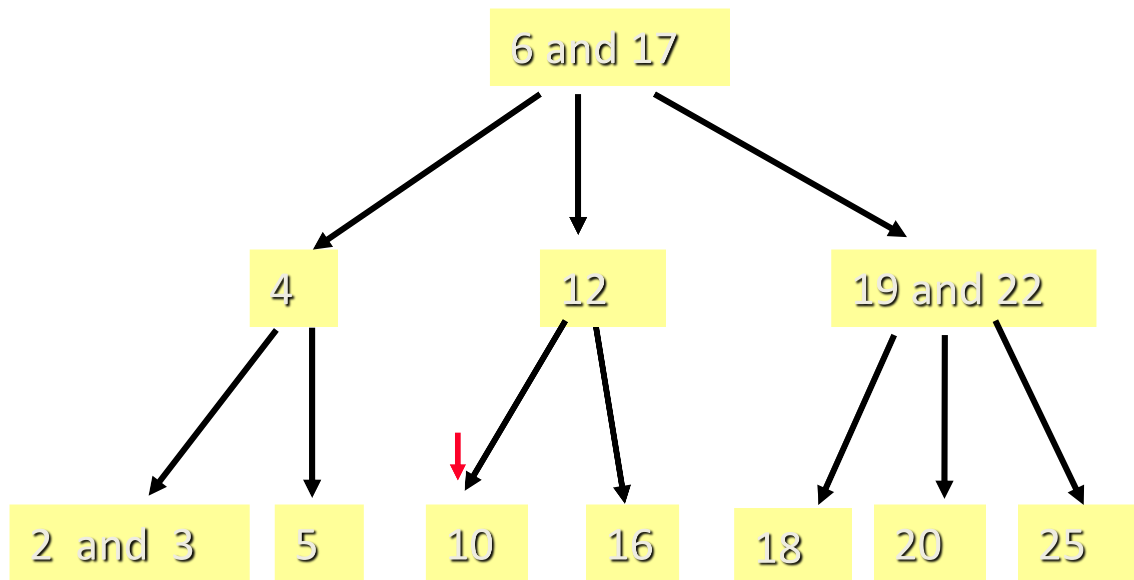
    return 1;

  else if (no children)

    return 0;

  else

    return subset[i]->count (target);



i = 0

data[i] is target !

Insert a Item into a B-Tree

- Prototype:
  - bool insert(const Item& entry);

- Post-condition:
  - If an equal entry was already in the set, the set is unchanged and the return value is false.
  - Otherwise, entry was added to the set and the return value is true.

Insert an Item in a B-Tree

Start at the root.

1) locate i so that !(data[i]<entry)

2) If (data[i] is entry)

    return false; // no work!

  else if (no children)

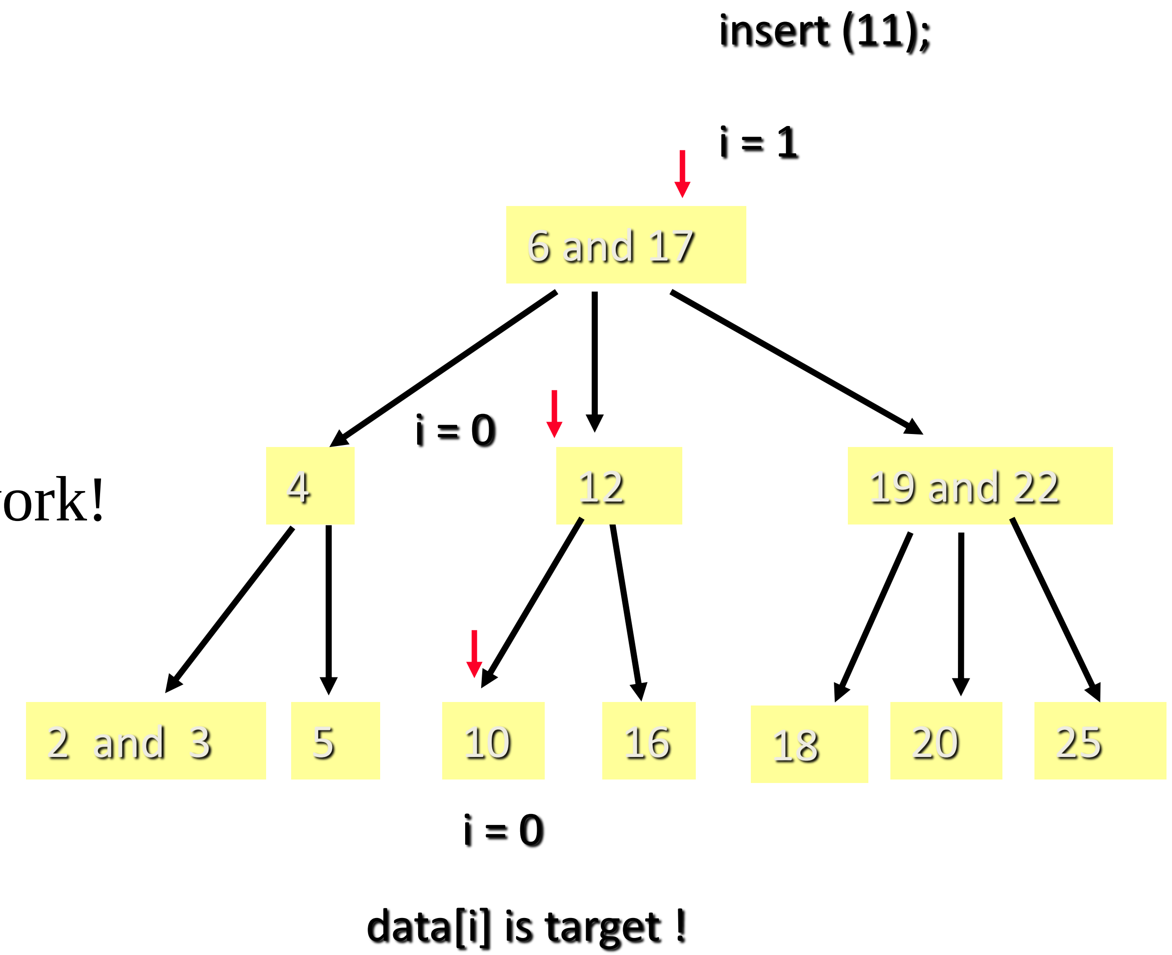
    insert entry at i;

    return true;

  else

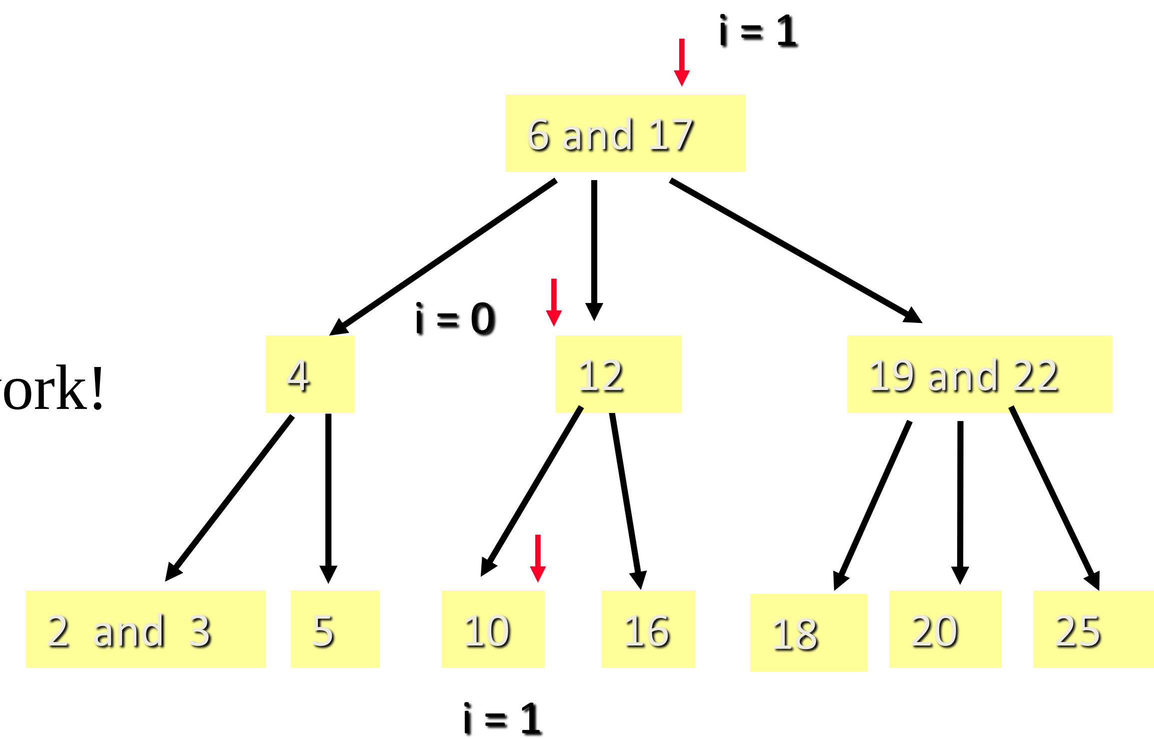
    return

   subset[i]->insert (entry);



Insert an Item in a B-Tree

insert (11); // MIN = 1 -> MAX = 2



data[0] < entry !

Start at the root.

1) locate i so that !(data[i]<entry)
2) If (data[i] is entry)

    return false; // no work!

  else if (no children)

    insert entry at i;

    return true;

  else

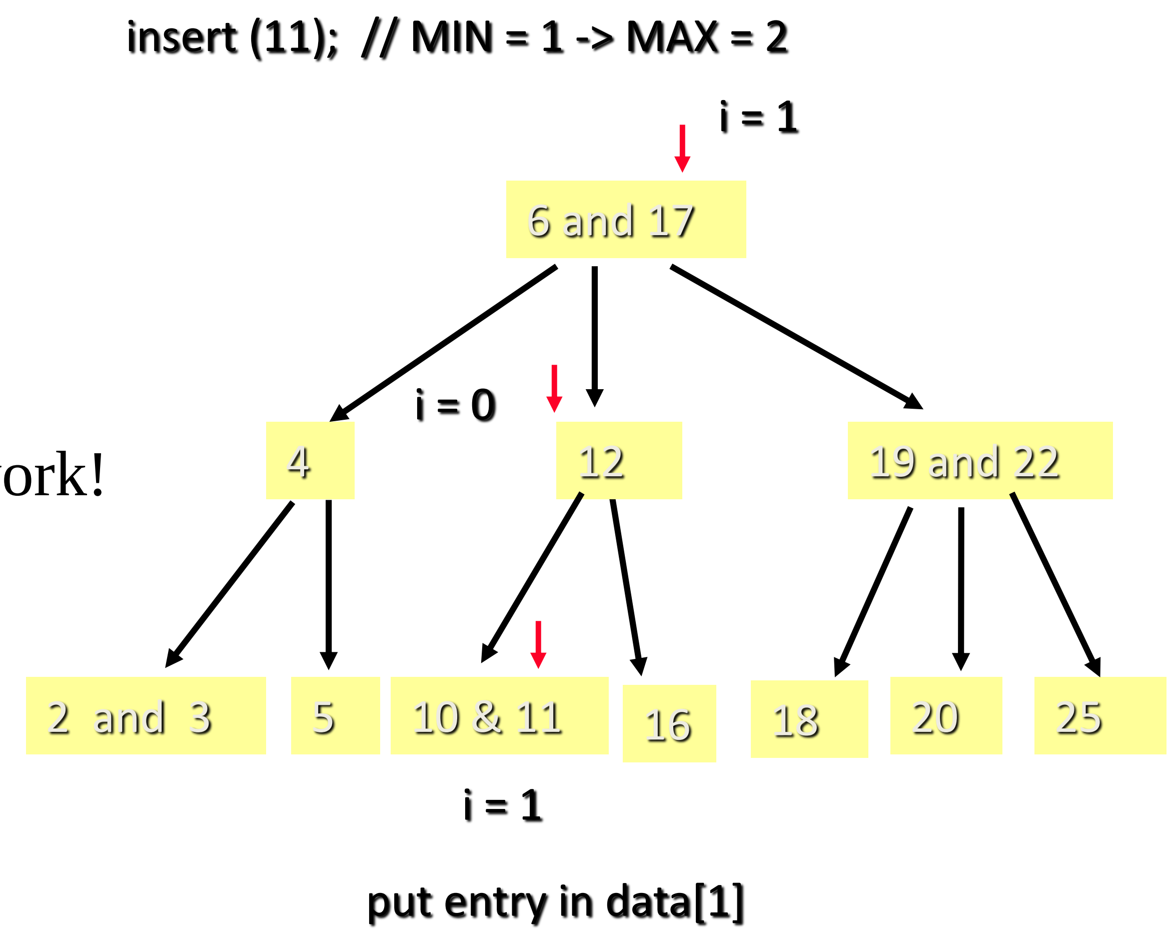
    return

   subset[i]->insert (entry);

Insert an Item in a B-Tree

Start at the root.

1) locate i so that !(data[i]<entry)
2) If (data[i] is entry)
        return false; // no work!
    else if (no children)
        insert entry at i;
        return true;
    else
        return
        subset[i]->insert (entry);



Inserting an Item into a B-Tree

- What if the node already have MAXIMUM number of items?
- Solution – loose insertion
  - A loose insert may results in MAX +1 entries in the root of a subset
  - Two steps to fix the problem:
    - ✓ fix it – but the problem may move to the root of the set
    - ✓ fix the root of the set

Insert an Item in a B-Tree

Start at the root.

1) locate i so that !(data[i]<entry)
2) If (data[i] is entry)

        return false; // no work!

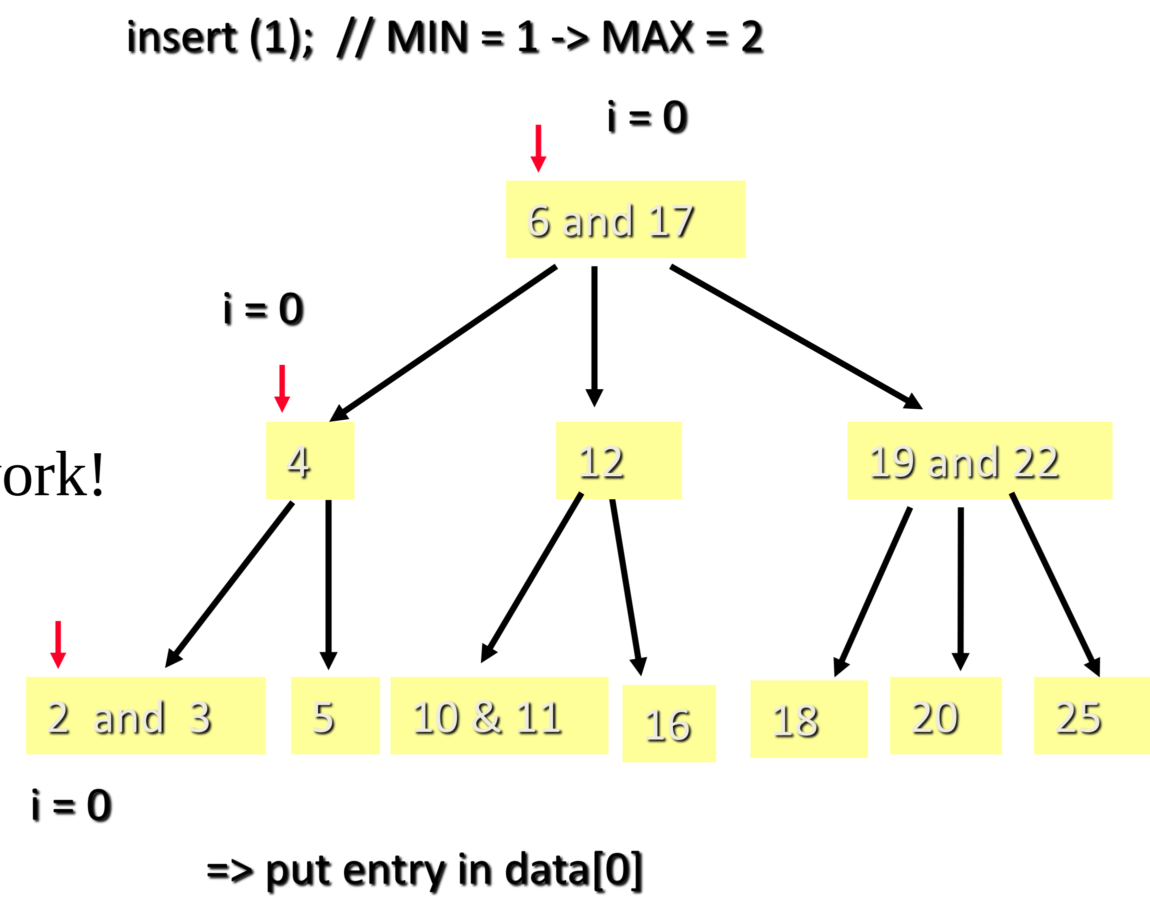
    else if (no children)

        insert entry at i;

        return true;

    else

        return subset[i]->insert (entry);



Insert an Item in a B-Tree

insert (1); // MIN = 1 -> MAX = 2

Start at the root.

1) locate i so that !(data[i]<entry)
2) If (data[i] is entry)

    return false; // no work!

  else if (no children)

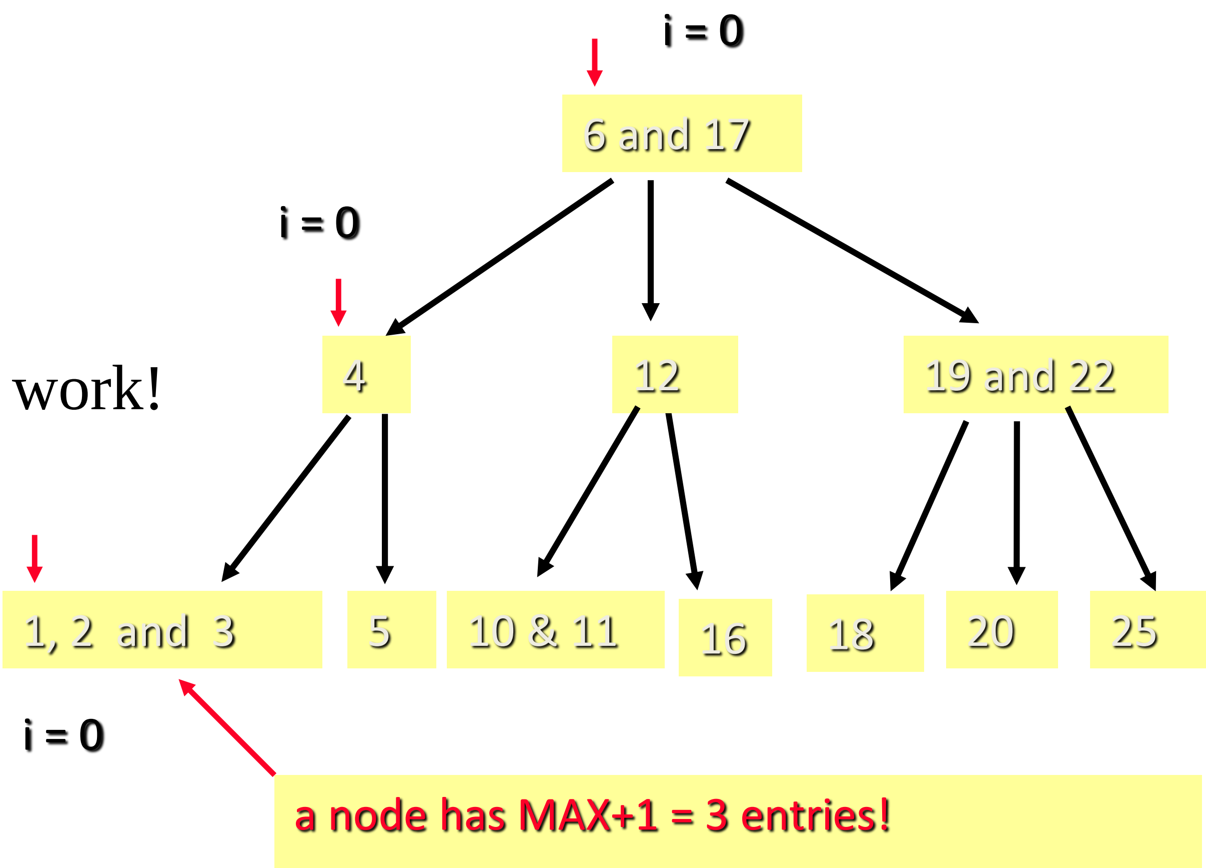
    insert entry at i;

    return true;

  else

    return

    subset[i]->insert (entry);

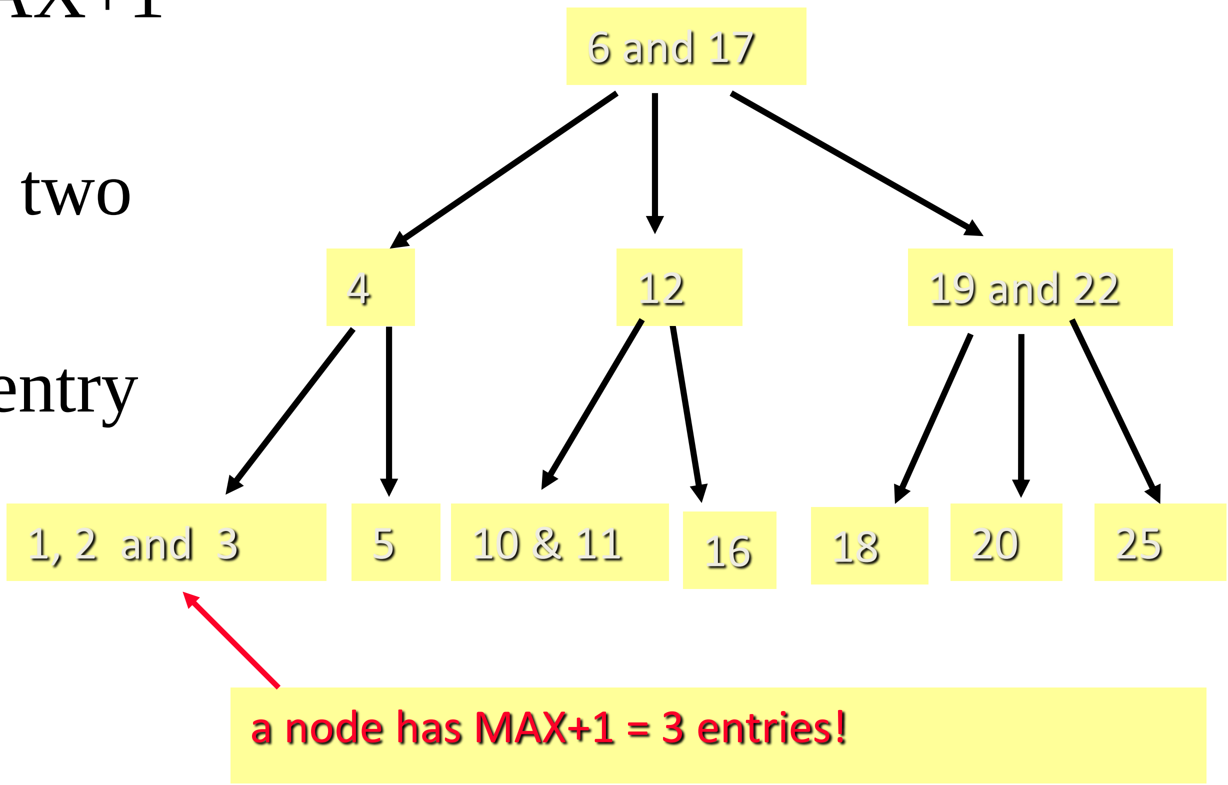


Insert an Item in a B-Tree

insert (1); // MIN = 1 -> MAX = 2

Fix the node with MAX+1 entries

- ☆ split the node into two from the middle
- ☆ move the middle entry up

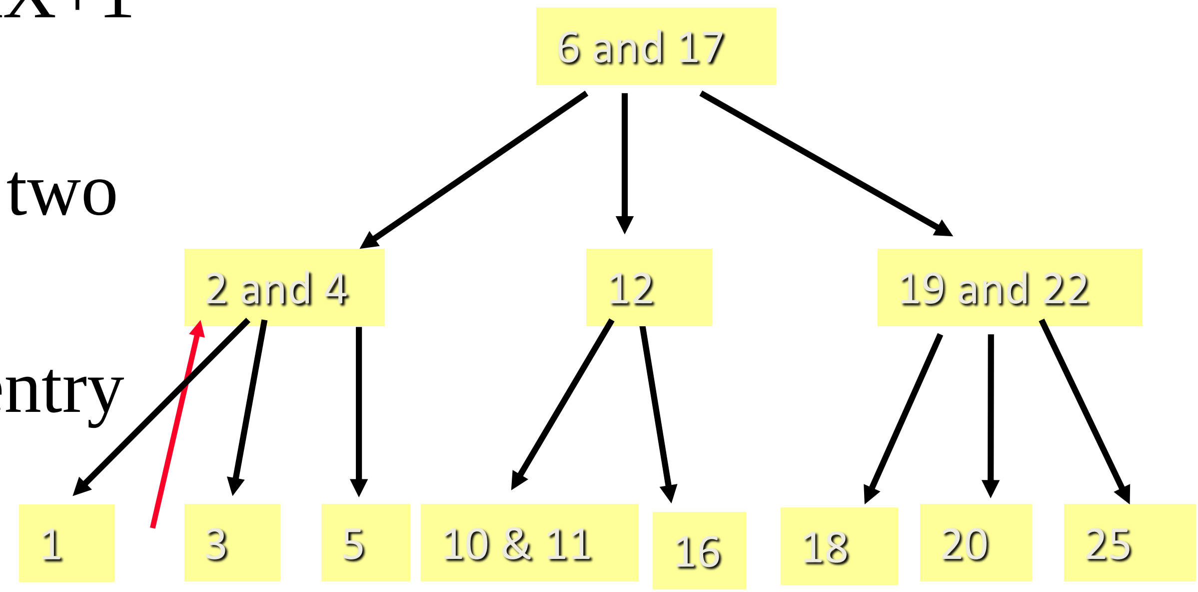


Insert an Item in a B-Tree

insert (1); // MIN = 1 -> MAX = 2

Fix the node with MAX+1 entries

- ☆ split the node into two from the middle
- ☆ move the middle entry up



Note: This shall be done recursively... the recursive function returns the middle entry to the root of the subset.

Erasing an Item from a B-Tree

- Prototype:
  - std::size_t erase(const Item& target);
- Post-Condition:
  - If target was in the set, then it has been removed from the set and the return value is 1.
  - Otherwise the set is unchanged and the return value is zero.

Erasing an Item from a B-Tree

- Similarly, after "loose erase", the root of a subset may just have MINIMUM –1 entries
- Solution
  - Fix the **shortage** of the subset root – but this may move the problem to the root of the entire set
  - Fix the **root** of the entire set (tree)

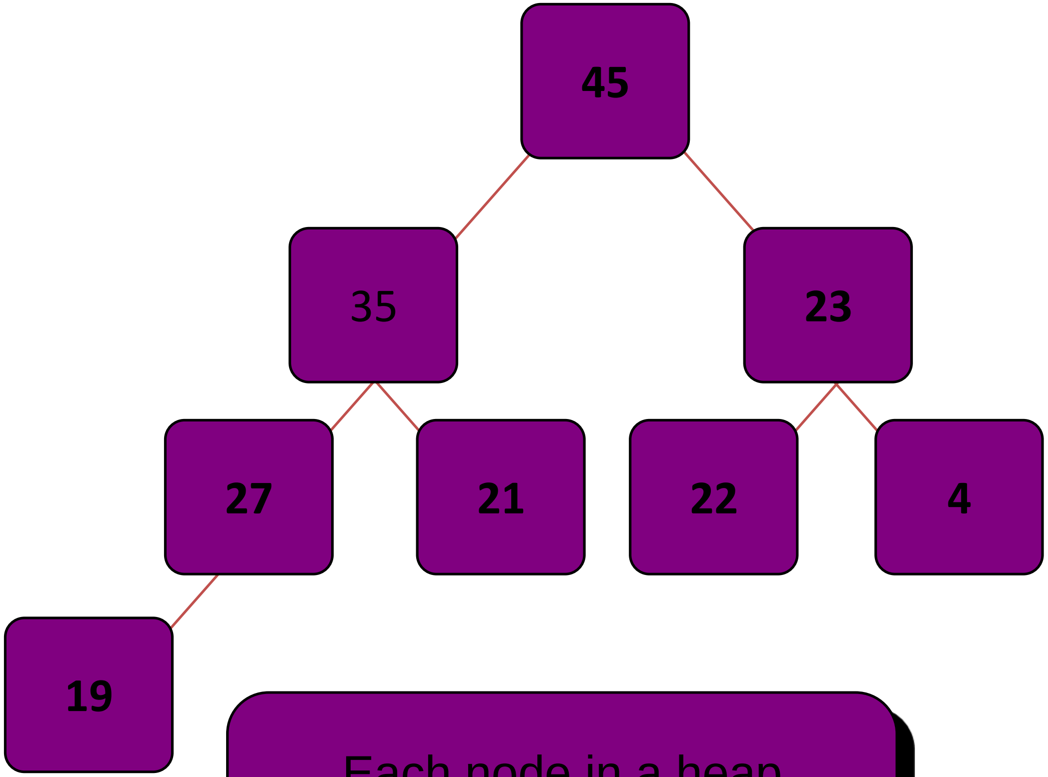
HEAPS AND PRIORITY QUEUES

Topics

- Heap Definition
- Heap Applications
  - priority queues (chapter 8), sorting (chapter 13)
- Two Heap Operations – add, remove
  - reheapification upward and downward
  - why is a heap good for implementing a priority queue?
- Heap Implementation
  - using binary_tree_node class
  - using fixed size or dynamic arrays

Heaps

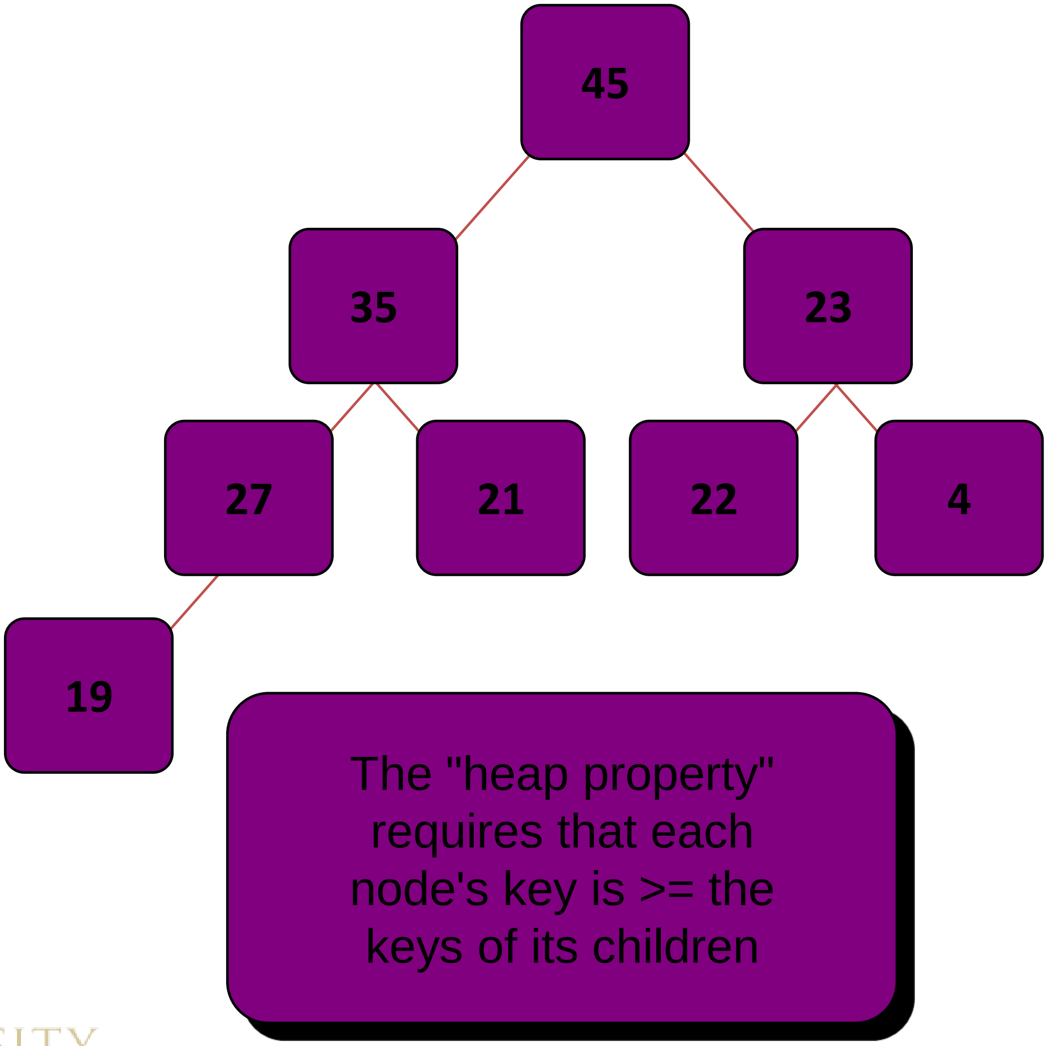
A heap is a **certain** kind of complete binary tree.



Each node in a heap contains a key that can be compared to other nodes' keys.

Heaps

A heap is a **certain** kind of complete binary tree.



What it is not: It is not a BST

- In a binary search tree, the entries of the nodes can be compared with a strict weak ordering. Two rules are followed for every node n:
  - The entry in node n is NEVER *less than* an entry in its left subtree
  - The entry in the node n is *less than* every entry in its right subtree.
- BST is not necessarily a complete tree

What it is: Heap Definition

- A heap is a binary tree where the entries of the nodes can be compared with the *less than* operator of a strict weak ordering. In addition, two rules are followed:
  - The entry contained by the node is NEVER *less than* the entries of the node's children
  - The tree is a COMPLETE tree.
- Q: where is the largest entry?

Application : Priority Queues

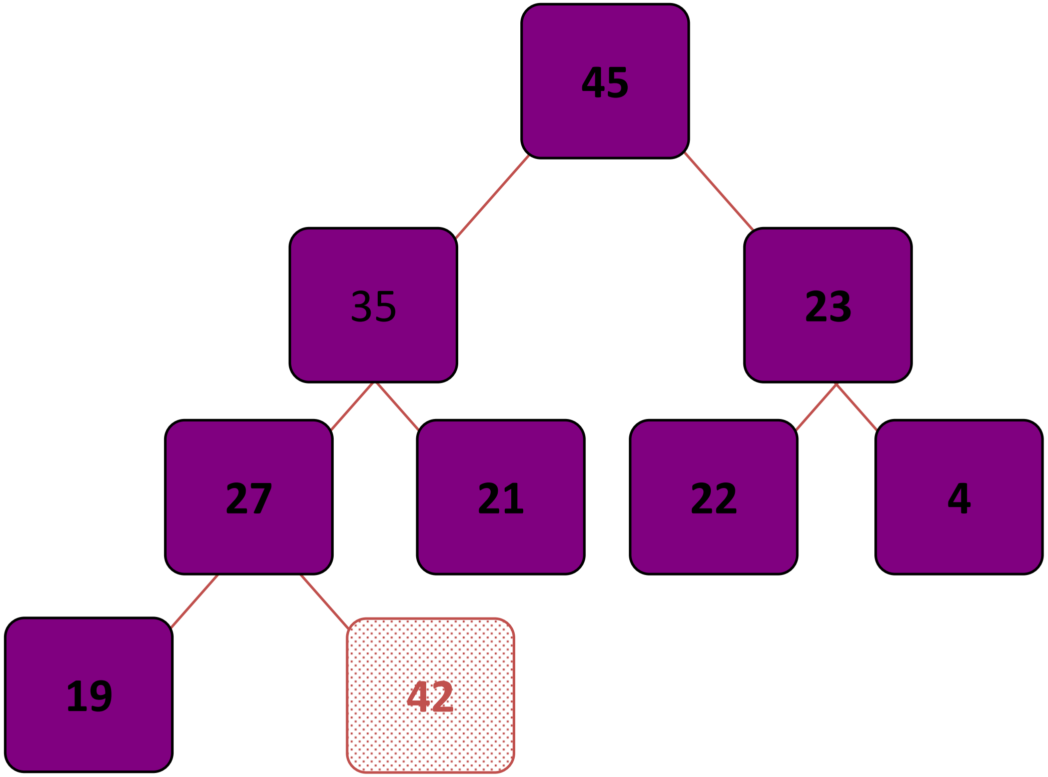
- A priority queue is a container class that allows entries to be retrieved according to some specific priority levels
  - The highest priority entry is removed first
  - If there are several entries with equally high priorities, then the priority queue's implementation determines which will come out first (e.g. FIFO)
- Heap is suitable for a priority queue

The Priority Queue ADT with Heaps

- The entry with the highest priority is always at the root node
- Focus on two priority queue operations
  - adding a new entry
  - remove the entry with the highest priority
- In both cases, we must ensure the tree structure remains to be a heap

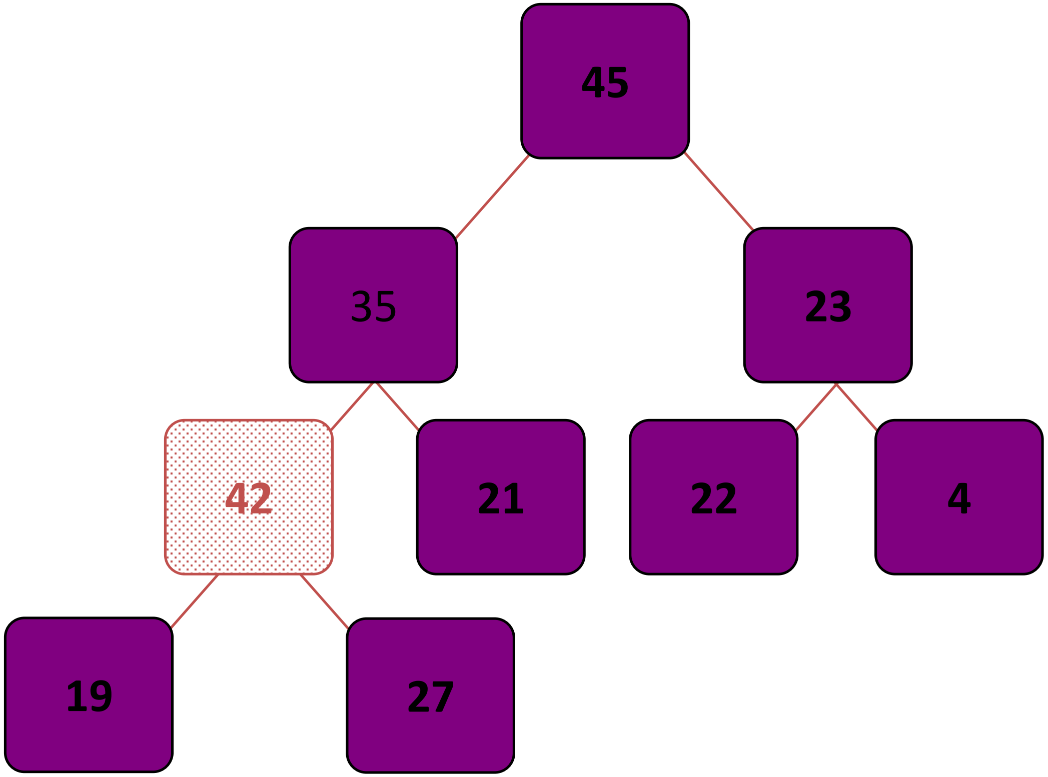
Adding a Node to a Heap

- Put the new node in the next available spot.
- Push the new node upward, swapping with its parent until the new node reaches an acceptable location.



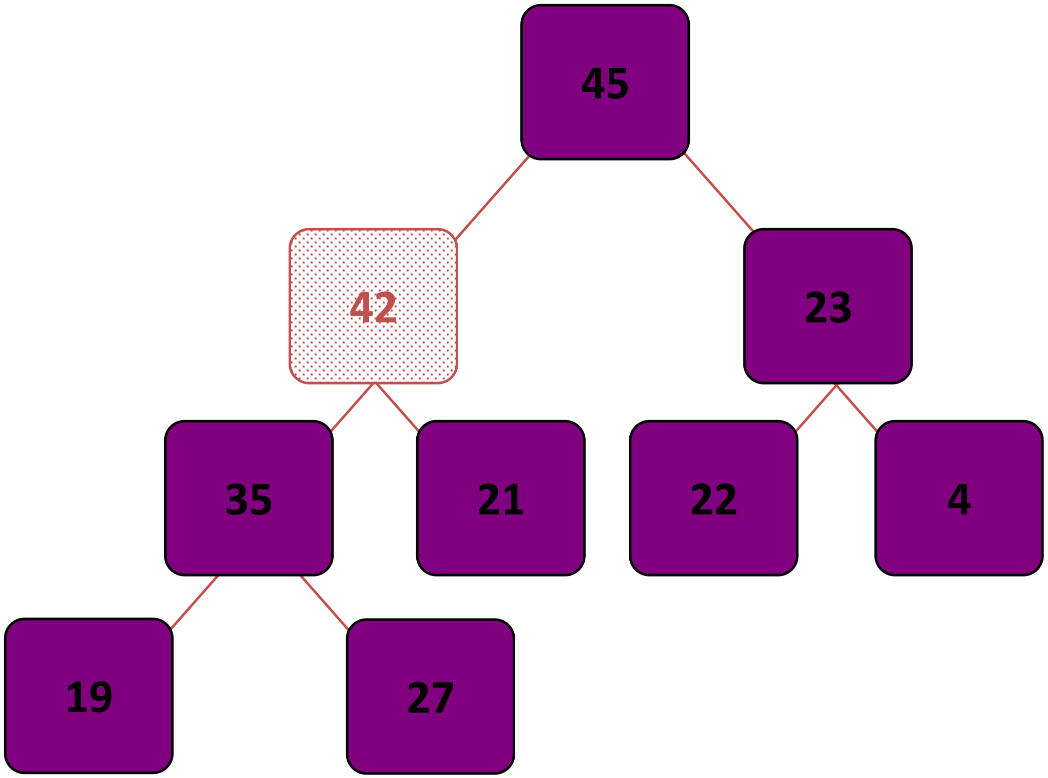
Adding a Node to a Heap

- Put the new node in the next available spot.
- Push the new node upward, swapping with its parent until the new node reaches an acceptable location.



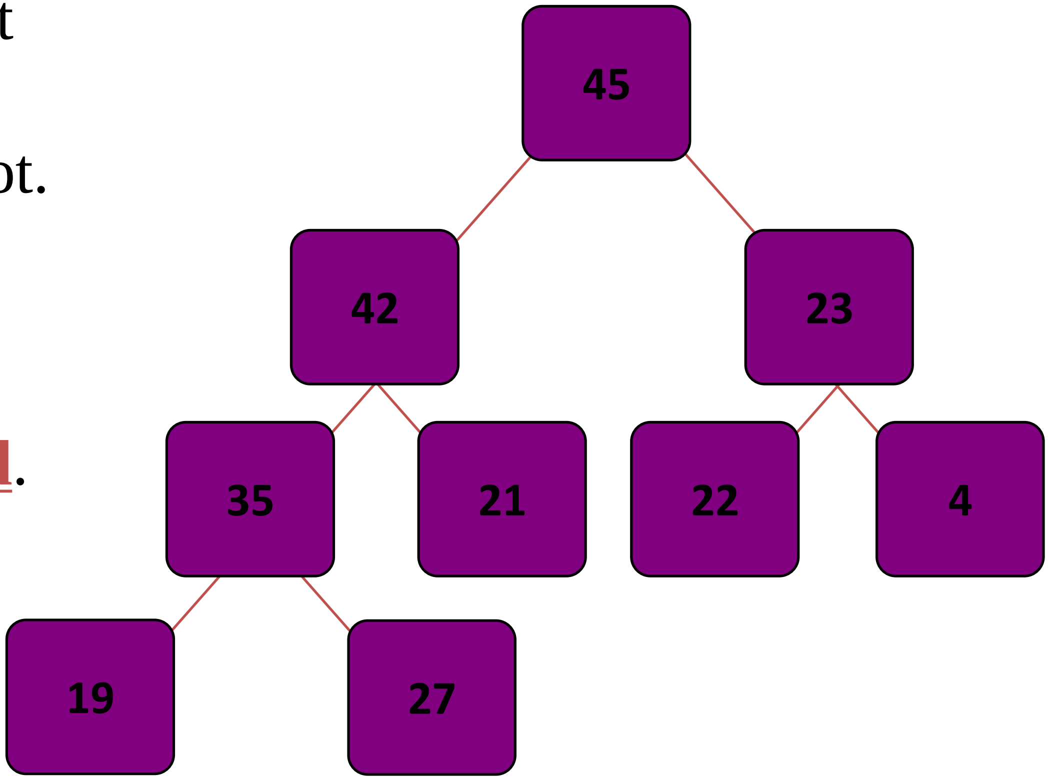
Adding a Node to a Heap

- Put the new node in the next available spot.
- Push the new node upward, swapping with its parent until the new node reaches an acceptable location.



Adding a Node to a Heap

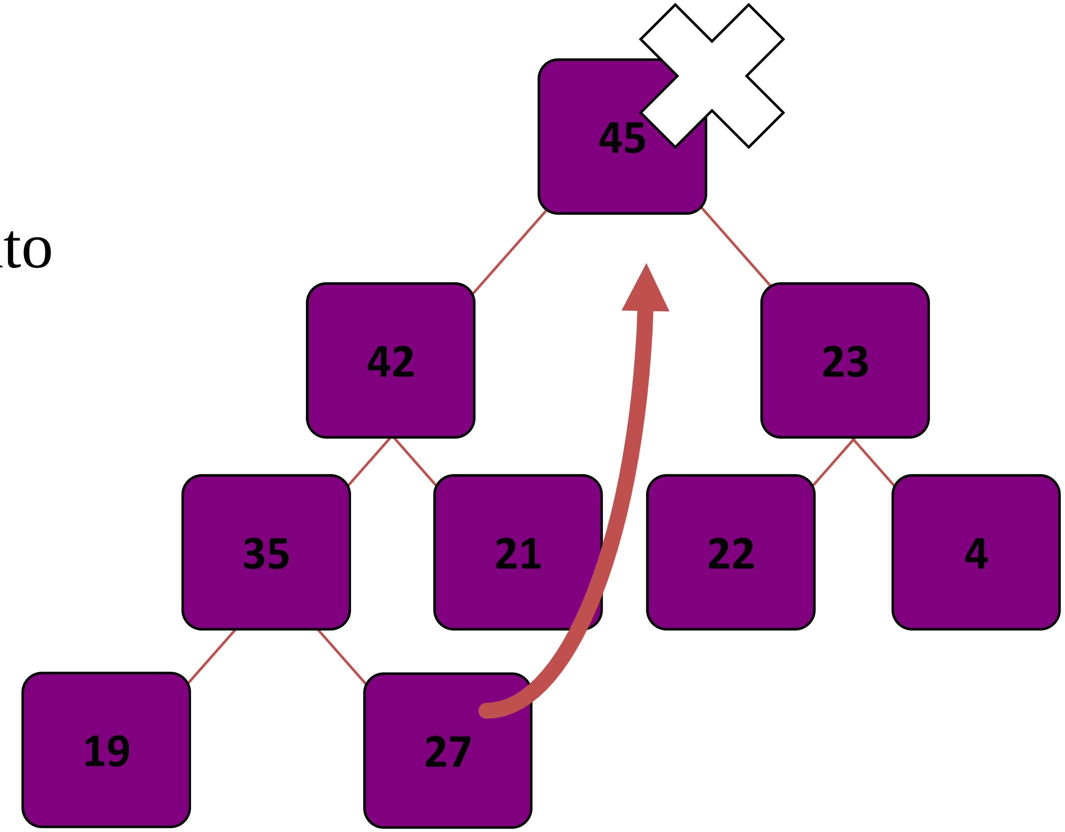
- The parent has a key that is >= new node, or
- The node reaches the root.
- The process of pushing the new node upward is called **reheapification upward**.



Note: we need to easily go from child to parent as well as parent to child.

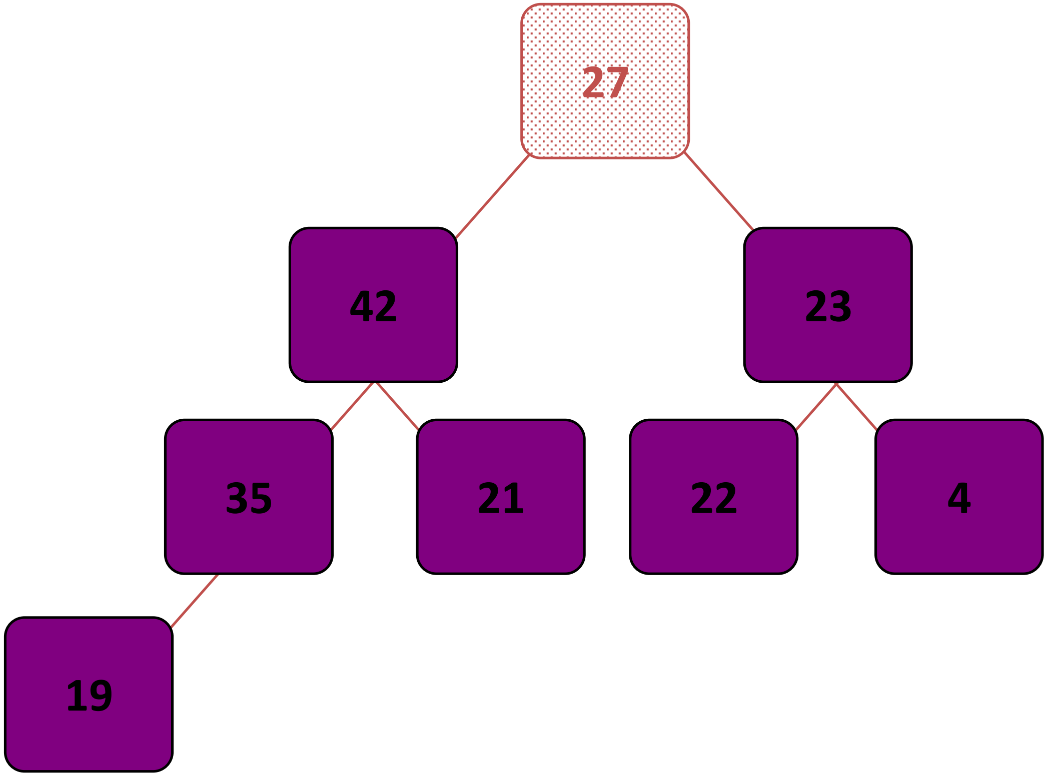
Removing the Top of a Heap

- Move the last node onto the root.



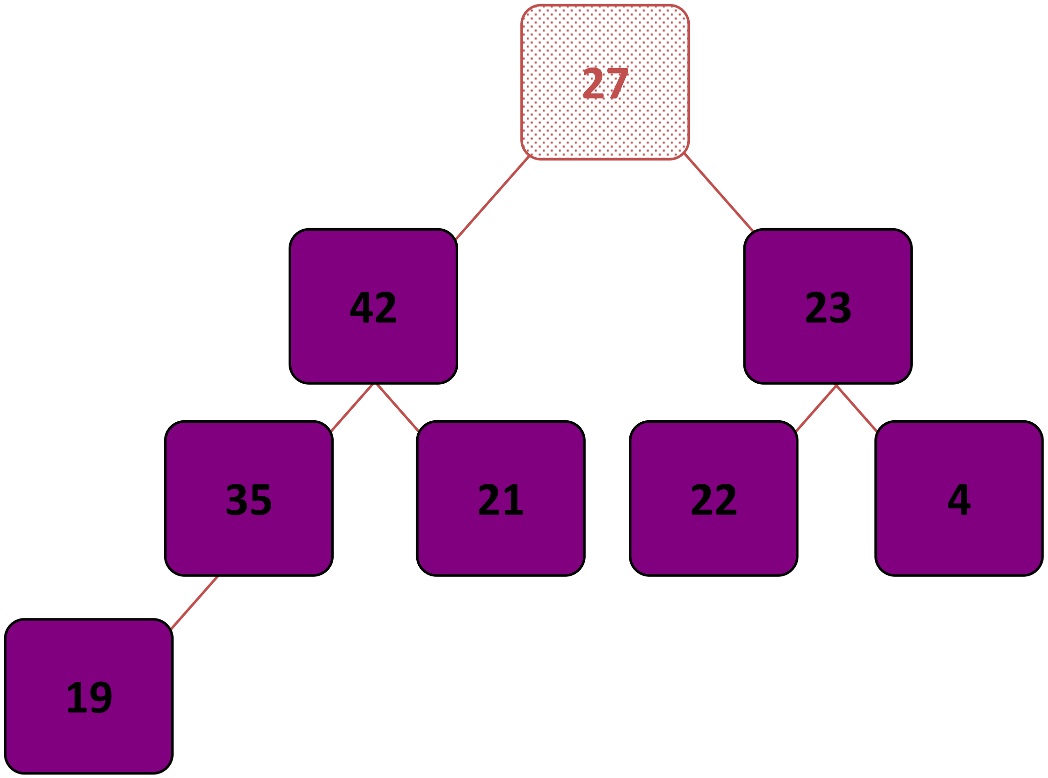
Removing the Top of a Heap

- Move the last node onto the root.



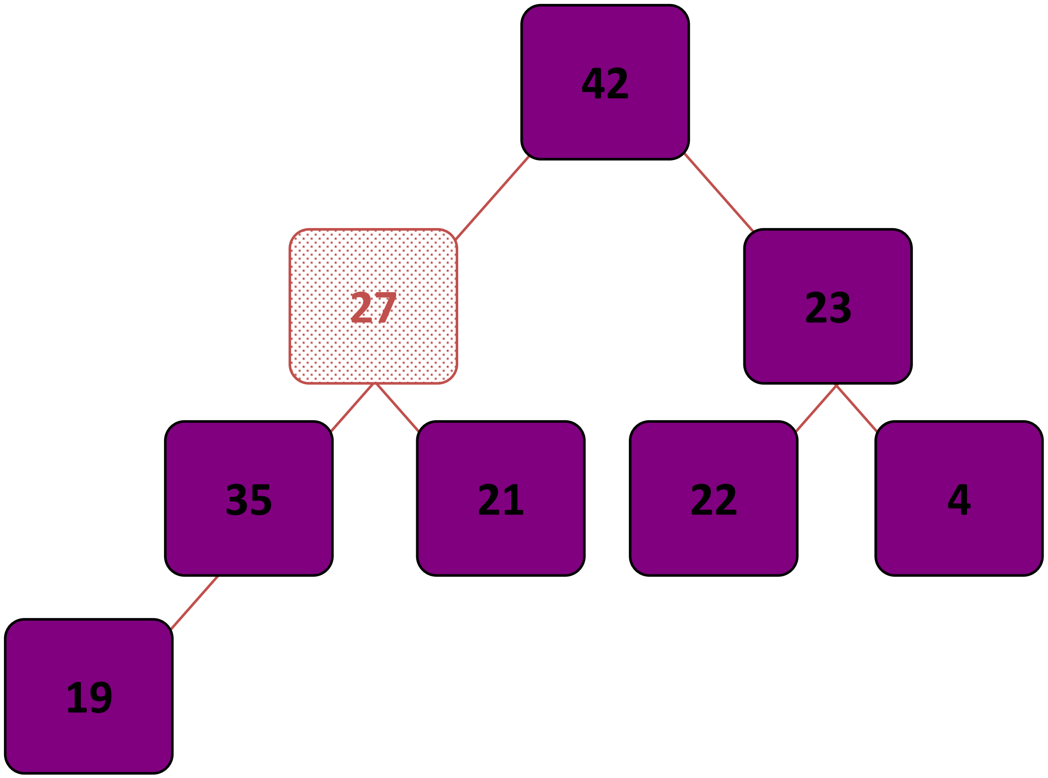
Removing the Top of a Heap

- Move the last node onto the root.
- Push the out-of-place node downward, swapping with its larger child until the new node reaches an acceptable location.



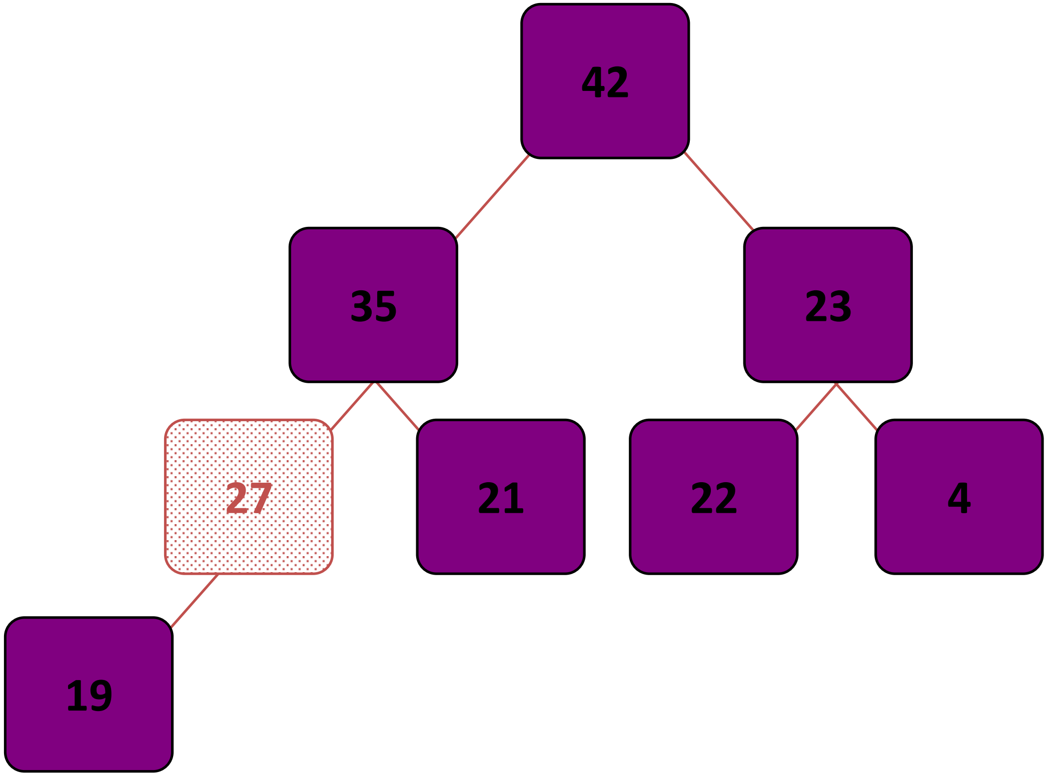
Removing the Top of a Heap

- Move the last node onto the root.
- Push the out-of-place node downward, swapping with its larger child until the new node reaches an acceptable location.



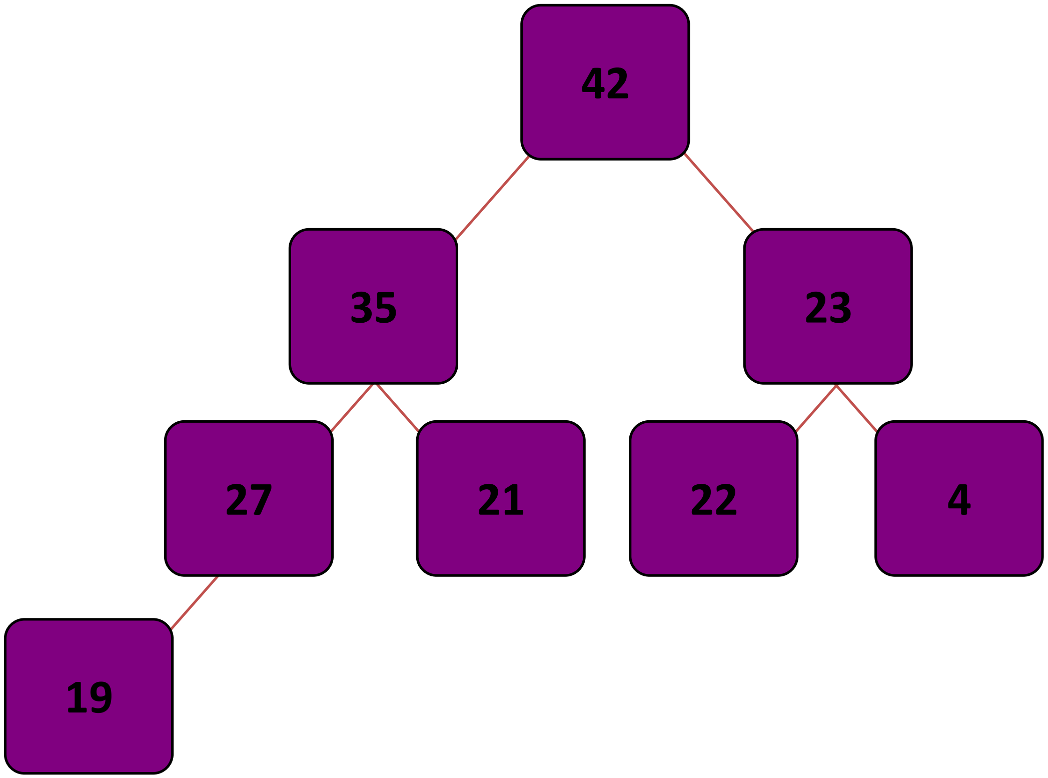
Removing the Top of a Heap

- Move the last node onto the root.
- Push the out-of-place node downward, swapping with its larger child until the new node reaches an acceptable location.



Removing the Top of a Heap

- The children all have keys <= the out-of-place node, or
- The node reaches the leaf.
- The process of pushing the new node downward is called **reheapification downward**.

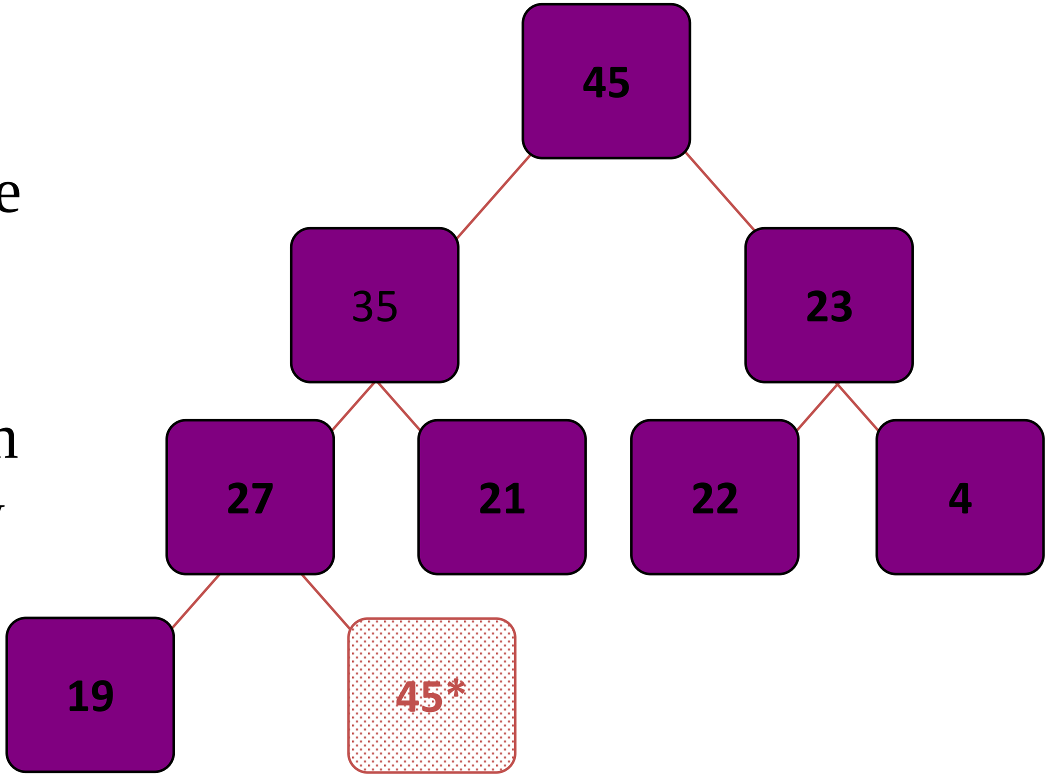


Priority Queues Revisited

- A priority queue is a container class that allows entries to be retrieved according to some specific priority levels
  - The highest priority entry is removed first
  - **If there are several entries with equally high priorities, then the priority queue's implementation determines which will come out first (e.g. FIFO)**
- Heap is suitable for a priority queue

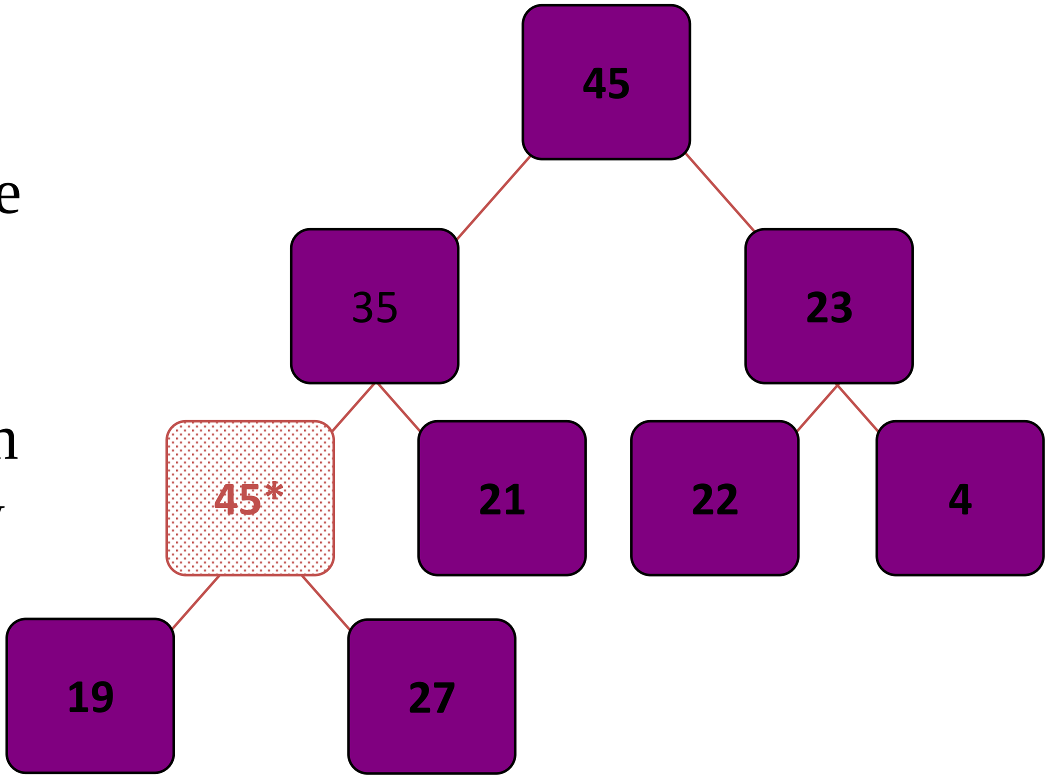
Adding a Node: same priority

- Put the new node in the next available spot.
- Push the new node upward, swapping with its parent until the new node reaches an acceptable location.



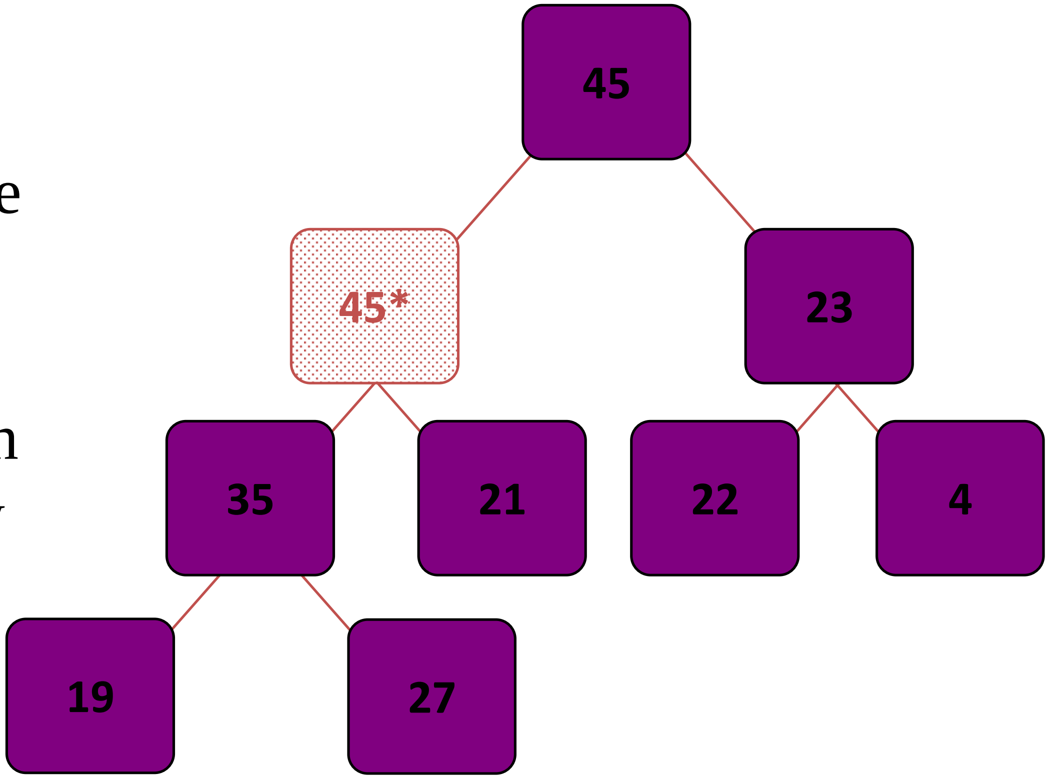
Adding a Node : same priority

- Put the new node in the next available spot.
- Push the new node upward, swapping with its parent until the new node reaches an acceptable location.



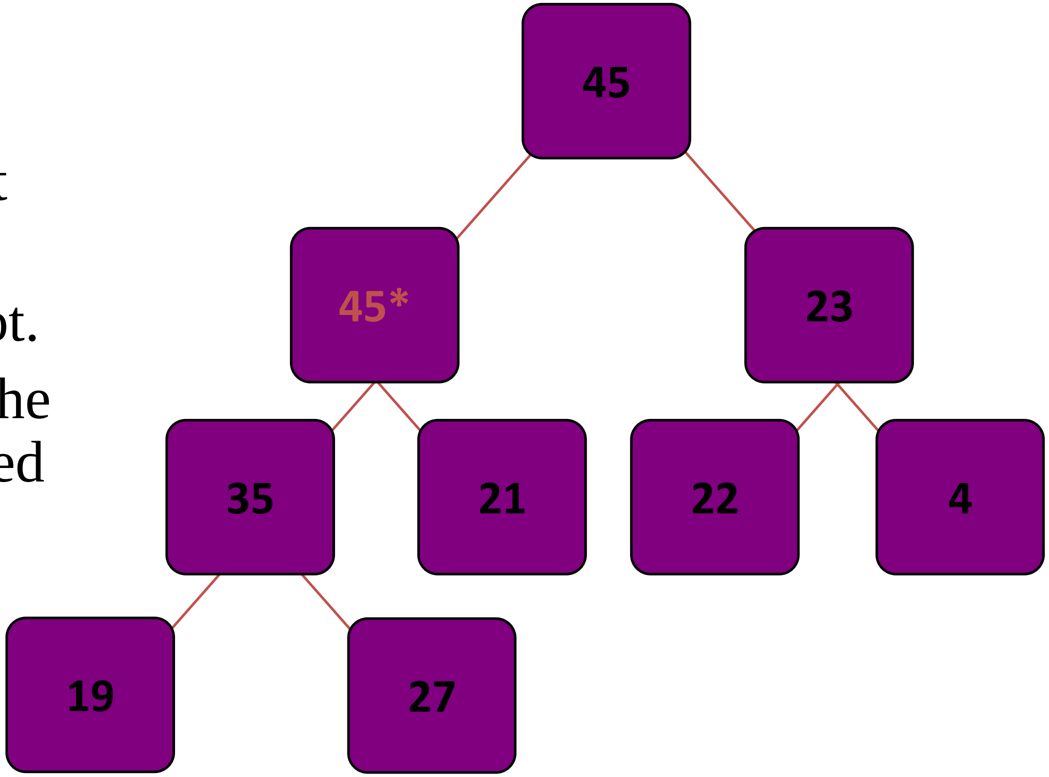
Adding a Node : same priority

- Put the new node in the next available spot.
- Push the new node upward, swapping with its parent until the new node reaches an acceptable location.



Adding a Node : same priority

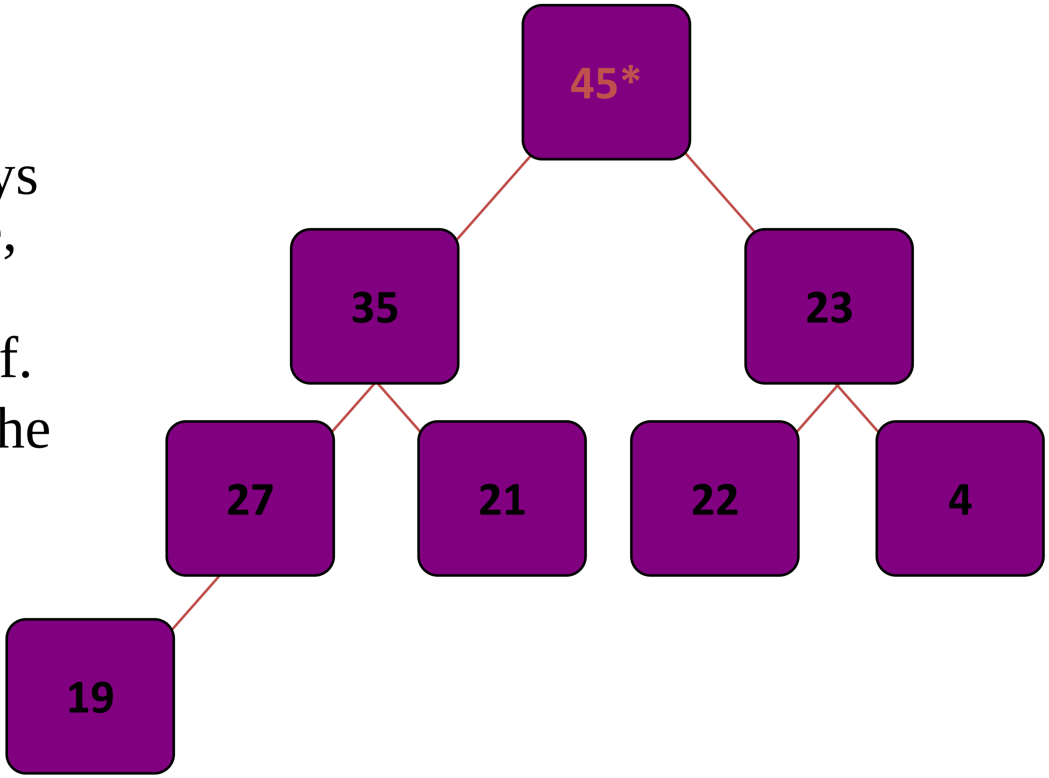
- The parent has a key that is >= new node, or
- The node reaches the root.
- The process of pushing the new node upward is called **reheapification upward**.



Note: Implementation determines which 45 will be in the root, and will come out first when popping.

Removing the Top of a Heap

- The children all have keys <= the out-of-place node, or
- The node reaches the leaf.
- The process of pushing the new node downward is called **reheapification downward**.



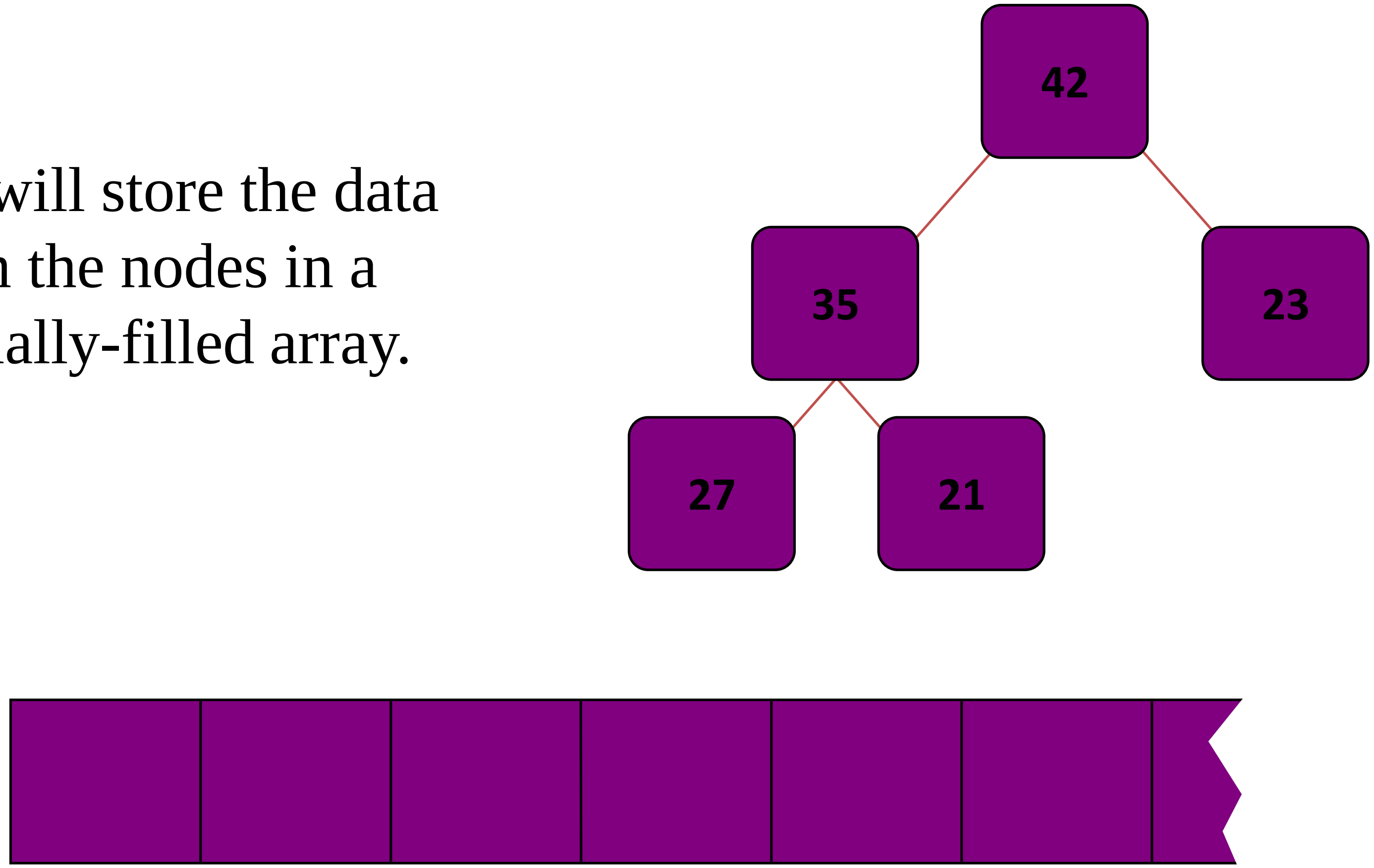
Note: Implementation determines which 45 will be in the root, and will come out first when popping.

Heap Implementation

- Use binary_tree_node class
  - node implementation is for a general binary tree
  - but we may need to have doubly linked node
- Use arrays
  - A heap is a complete binary tree
  - which can be implemented more easily with an array than with the node class
  - and do two-way links

Implementing a Heap

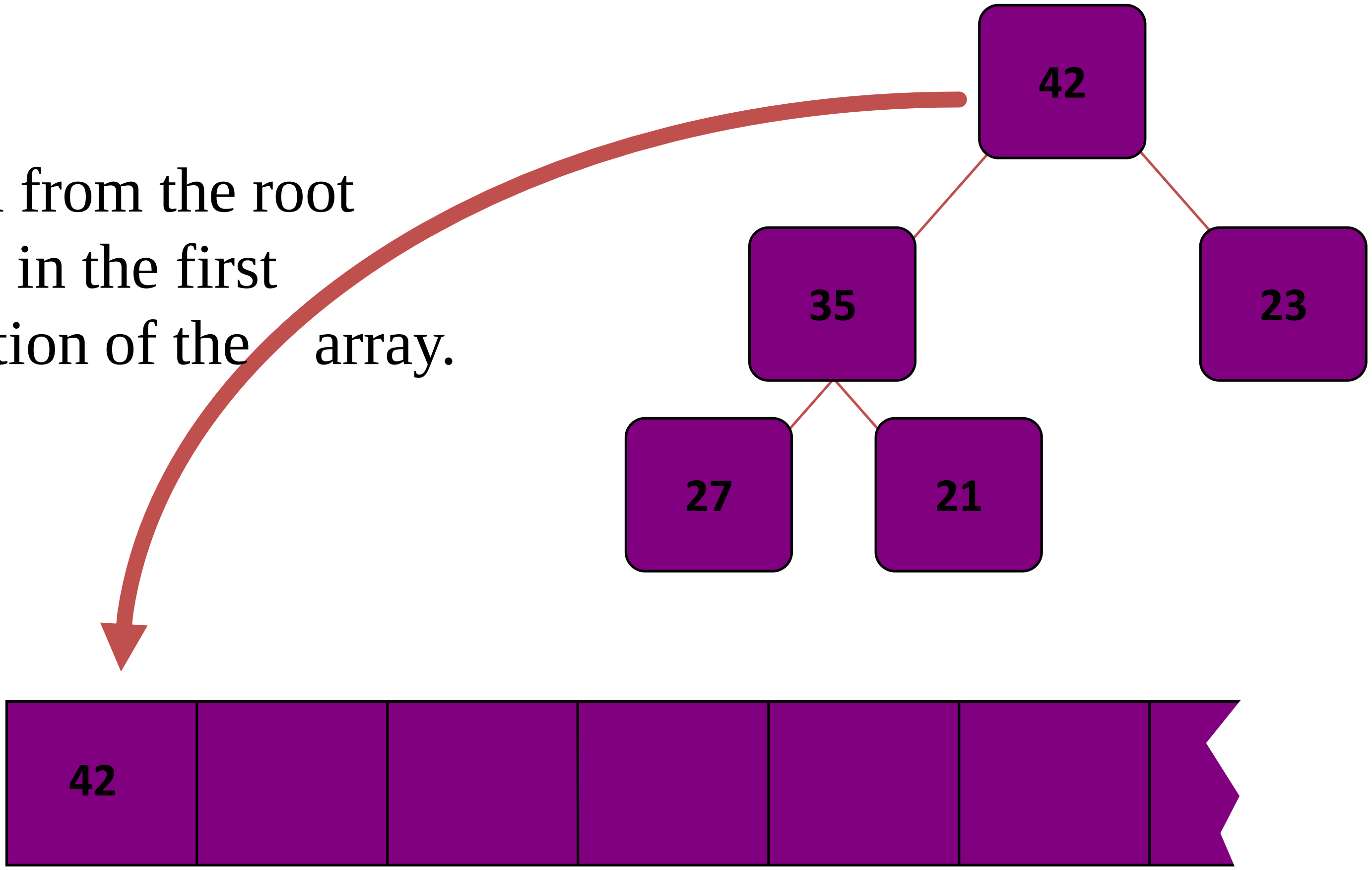
- We will store the data from the nodes in a partially-filled array.



An array of data

Implementing a Heap

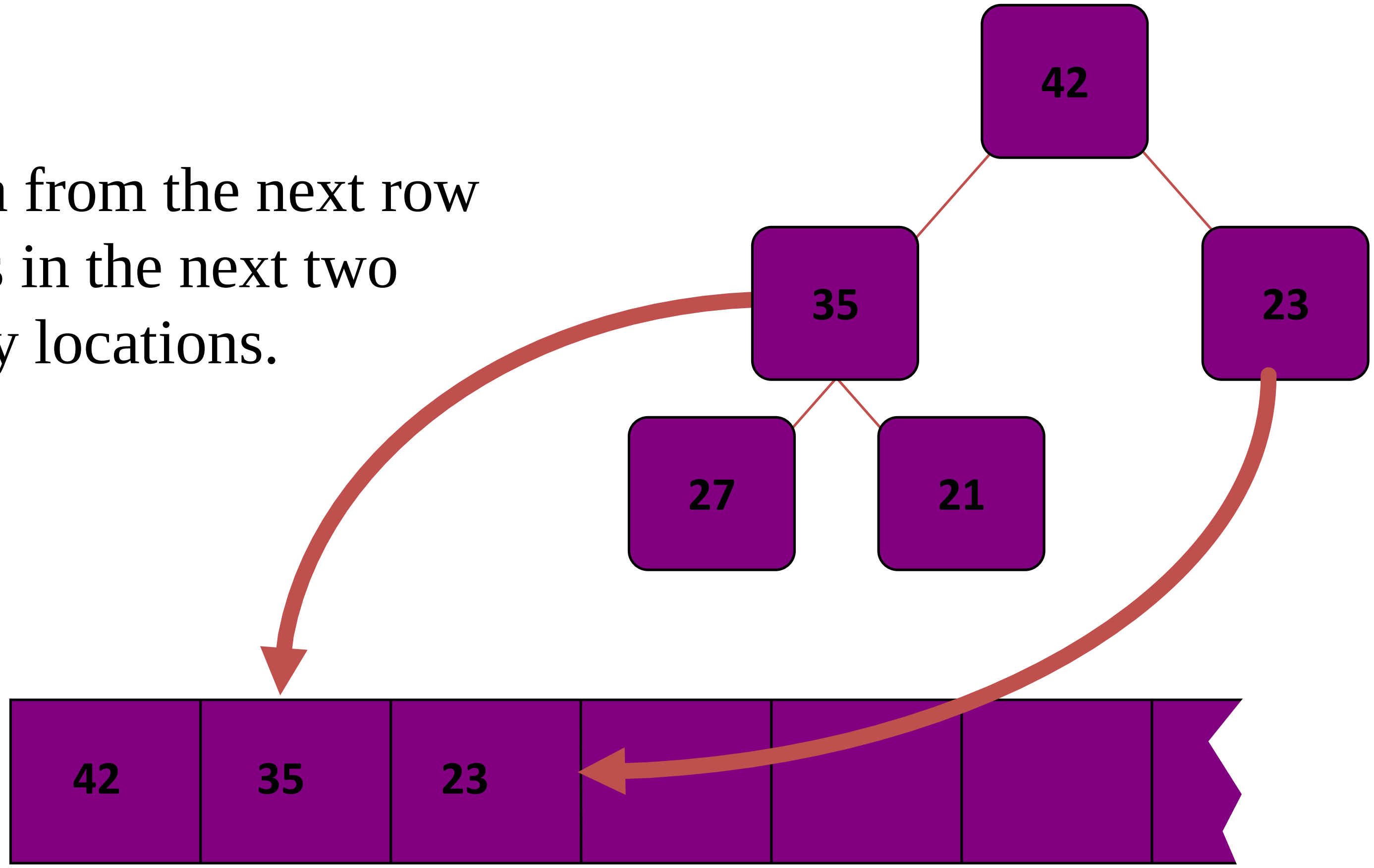
- Data from the root goes in the first location of the array.



An array of data

Implementing a Heap

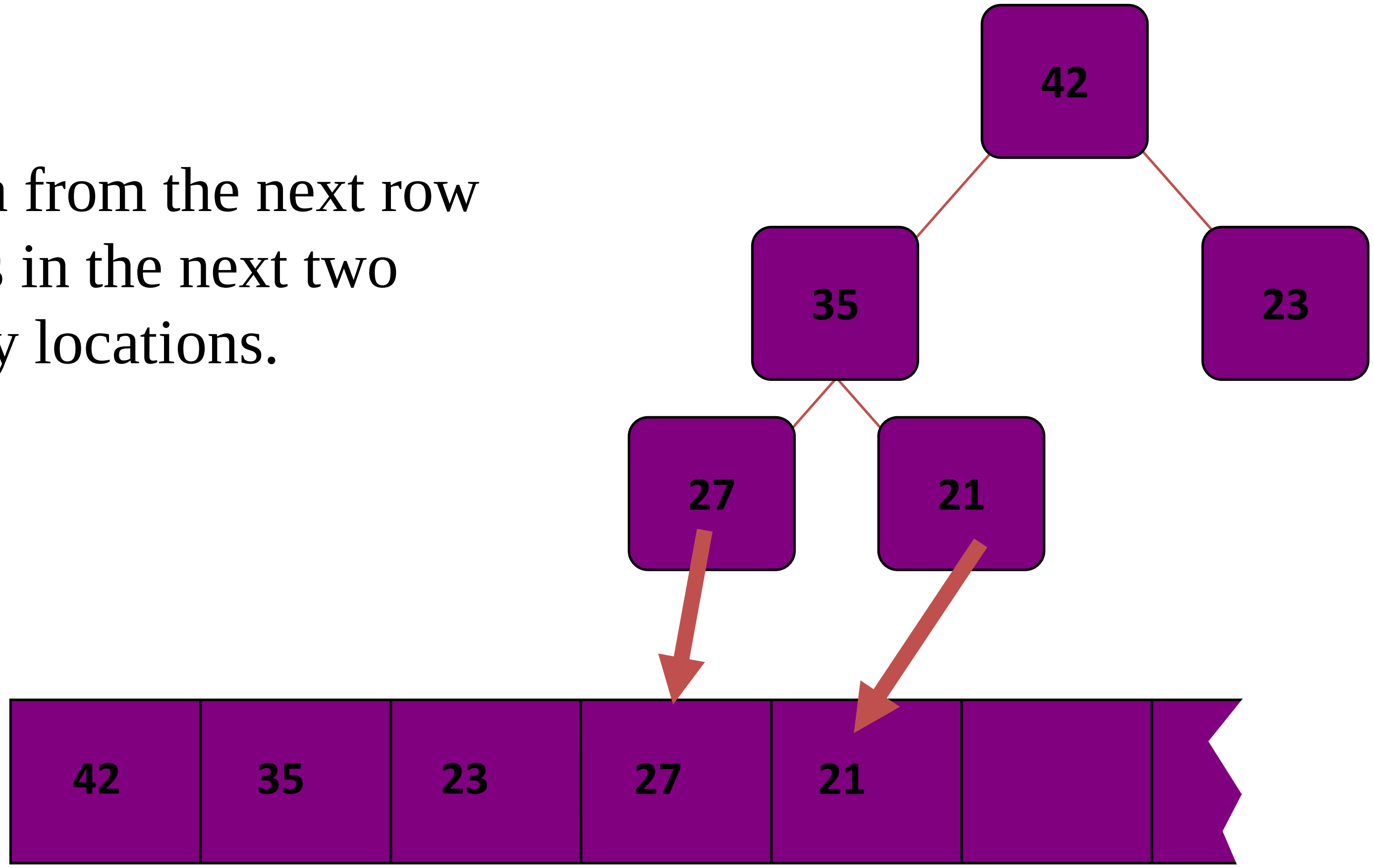
- Data from the next row goes in the next two array locations.



An array of data

Implementing a Heap

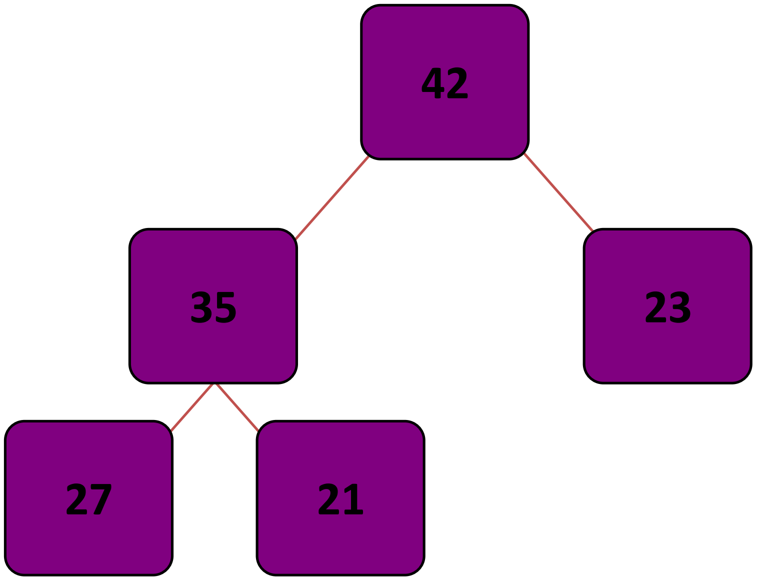
- Data from the next row goes in the next two array locations.

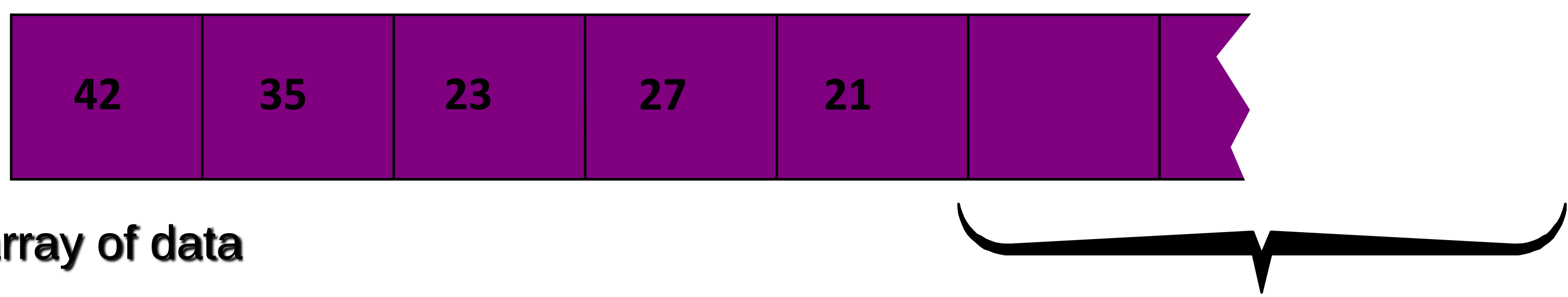


An array of data

Implementing a Heap

- Data from the next row goes in the next two array locations.



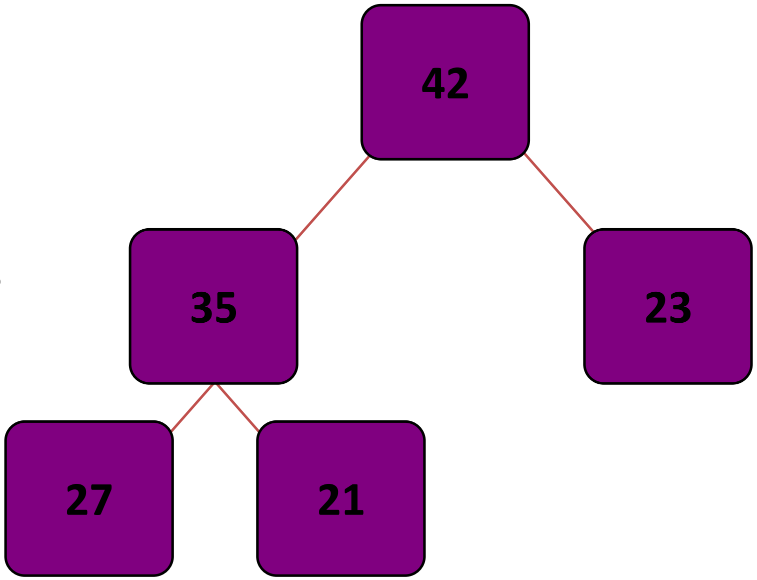


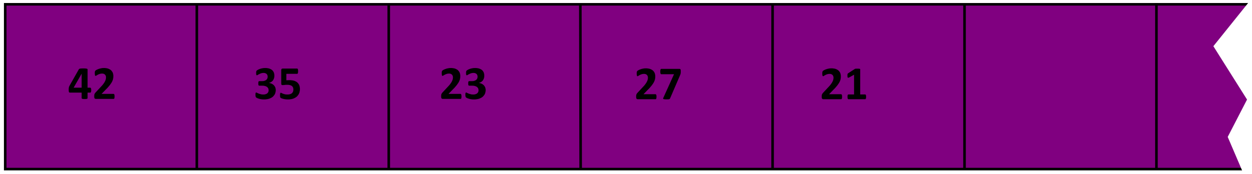
An array of data

We don't care what's in this part of the array.

Important Points about the Implementation

- The links between the tree's nodes are **not** actually stored as pointers, or in any other way.
- The only way we "know" that "the array is a tree" is from the way we manipulate the data.

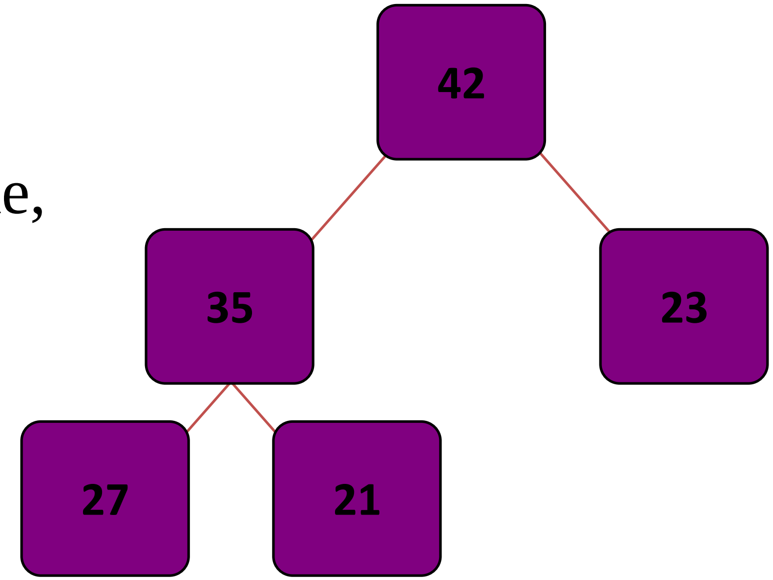


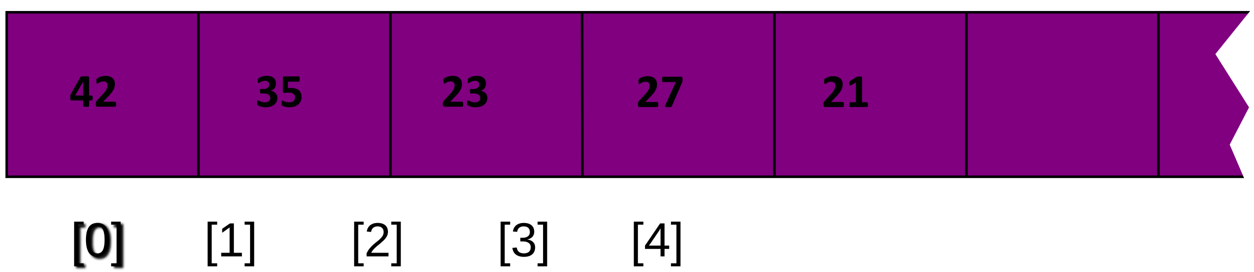


An array of data

Important Points about the Implementation

- If you know the index of a node, then it is easy to figure out the indexes of that node's parent and children.





Formulas for location of children and parents in an array representation

- Root at location [0]
- Parent of the node in [i] is at [(i-1)/2]
- Children of the node in [i] (if exist) is at [2i+1] and [2i+2]
- Test:
  - complete tree of 10, 000 nodes
  - parent of 4999 is at (4999-1)/2 = 2499
  - children of 4999 is at 9999 (V) and 10,000 (X)

TREES, LOGS AND TIME ANALYSIS

Topics

- Big-O Notation
- Worse Case Times for Tree Operations
- Time Analysis for BSTs
- Time Analysis for Heaps
- Logarithms and Logarithmic Algorithms

Big-O Notation

- The order of an algorithm generally is more important than the speed of the processor

Input size: n	O(log n)	O (n)	O (n^2)
# of stairs: n	$[\log_{10} n]+1$	3n	n^2+2n
10	2	30	120
100	3	300	10,200
1000	4	3000	1,000,2000

Worst-Case Times for Tree Operations

- The worst-case time complexity for the following are all O(d), where d = the depth of the tree:
  - Adding an entry in a BST, a heap or a B-tree;
  - Deleting an entry from a BST, a heap or a B-tree;
  - Searching for a specified entry in a BST or a B-tree.

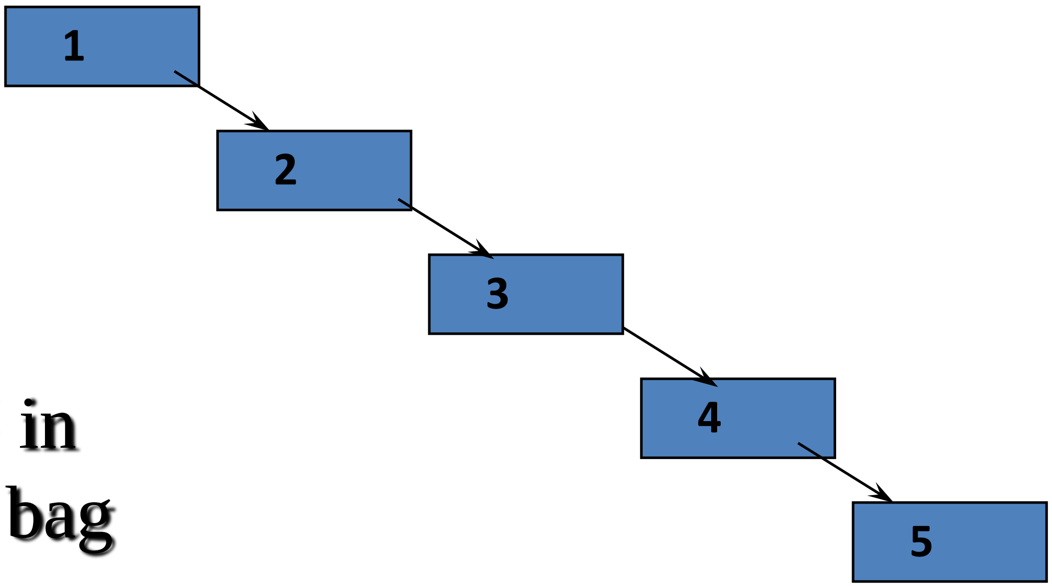
- This seems to be the end of our Big-O story...but

What's *d*, then?

- Time Analyses for these operations are more useful if they are given in term of the number of entries (n) instead of the tree's depth (d)
- Question:
  - What is the maximum depth for a tree with n entries?

Time Analysis for BSTs

- Maximum depth of a BST with n entires: n-1



- **An Example:**

  Insert 1, 2, 3,4,5 in that order into a bag using a BST

Worst-Case Times for BSTs

- Adding, deleting or searching for an entry in a BST with n entries is O(d), where d is the depth of the BST
- Since d is no more than n-1, the operations in the worst case is (n-1).
- Conclusion: the worst case time for the add, delete or search operation of a BST is O(n)

Time Analysis for Heaps

- A heap is a complete tree
- The minimum number of nodes needed for a heap to reach depth d is 2^d :
  - $= (1 + 2 + 4 + ... + 2^{d-1}) + 1$
  - The extra one at the end is required since there must be at least one entry in level n
- Question: how to add up the formula?

Time Analysis for Heaps

- A heap is a complete tree
- The minimum number of nodes needed for a heap to reach depth d is 2^d :
- The number of nodes n >= 2^d
- Use base 2 logarithms on both side
  - $\log_2 n >= \log_2 2^d = d$
  - Conclusion: $d <= \log_2 n$

Worst-Case Times for Heap Operations

- Adding or deleting an entry in a heap with n entries is O(d), where d is the depth of the tree
- Because d is no more than $\log_2 n$, we conclude that the operations are O(log n)

- Why we can omit the subscript 2 ?

Logarithms (log)

- Base 10: the number of digits in n is [$\log_{10} n$]+1
  - $10^0 = 1$, so that $\log_{10} 1 = 0$
  - $10^1 = 10$, so that $\log_{10} 10 = 1$
  - $10^{1.5} = 32+$, so that $\log_{10} 32 = 1.5$
  - $10^3 = 1000$, so that $\log_{10} 1000 = 3$
- Base 2:
  - $2^0 = 1$, so that $\log_2 1 = 0$
  - $2^1 = 2$, so that $\log_2 2 = 1$
  - $2^3 = 8$, so that $\log_2 8 = 3$
  - $2^5 = 32$, so that $\log_2 32 = 5$
  - $2^{10} =1024$, so that $\log_2 1024 = 10$

Logarithms (log)

- Base 10: the number of digits in n is $[\log_{10} n]+1$
  - $10^{1.5} = 32+$, so that $\log_{10} 32 = 1.5$
  - $10^{3} = 1000$, so that $\log_{10} 1000 = 3$
- Base 2:
  - $2^{3} = 8$, so that $\log_{2} 8 = 3$
  - $2^{5} = 32$, so that $\log_{2} 32 = 5$
- Relation: For any two bases, a and b, and a positive number n, we have
  - $\mathbf{\log_b n = (\log_b a) \log_a n} = \log_b a^{(\log_a n)}$
  - $\mathbf{\log_2 n} = (\log_2 10) \log_{10} n = (5/1.5) \log_{10} n = \mathbf{3.3 \log_{10} n}$

Logarithmic Algorithms

- Logarithmic algorithms are those with worst-case time O(log n), such as adding to and deleting from a heap
- For a logarithm algorithm, doubling the input size (n) will make the time increase by a fixed number of new operations
- Comparison of linear and logarithmic algorithms
  - n= m = 1 hour -> $\log_2 m \approx 6$ minutes
  - n=2m = 2 hour -> $\log_2 m + 1 \approx 7$ minutes
  - n=8m = 1 work day -> $\log_2 m + 3 \approx 9$ minutes
  - n=24m = 1 day&night -> $\log_2 m + 4.5 \approx 10.5$ minutes

Summary

- Big-O Notation :
  - Order of an algorithm versus input size (n)
- Worse Case Times for Tree Operations
  - O(d), d = depth of the tree
- Time Analysis for BSTs
  - worst case: O(n)
- Time Analysis for Heaps
  - worst case O(log n)
- Logarithms and Logarithmic Algorithms
  - doubling the input only makes time increase a fixed number