INFO 6205
Program Structure and Algorithms

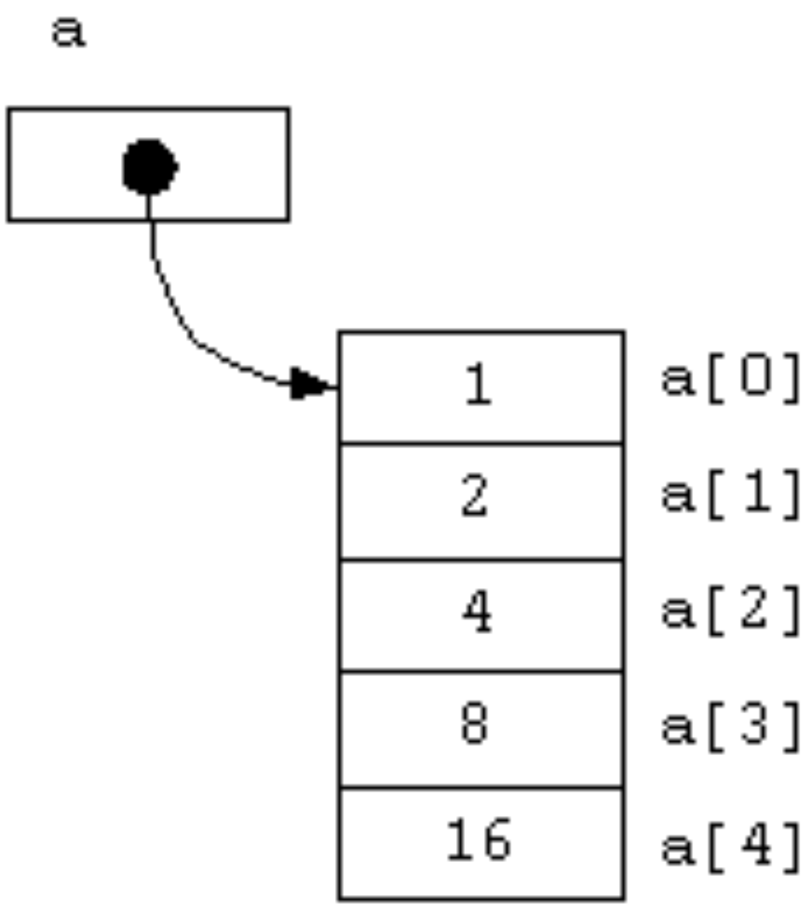
Nik Bear Brown

Data Structures

Topics

- Arrays
- Linked Lists
- Doubly linked
- Stacks
- Trees (Binary Trees, AVL (Adelson-Velskii and Landis) Trees, m-way trees , B+ trees)
- Hash Tables
- Heaps

Data Structures - Arrays



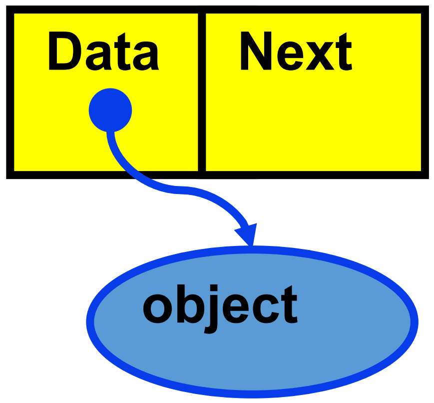
Array Limitations

- Arrays
  - Simple,
  - Fast

  but
  - Must specify size at construction time
  - Murphy's law
    - Construct an array with space for n
      - n = twice your estimate of largest collection
    - Tomorrow you'll need n+1
  - More flexible system?

Linked Lists

- Flexible space use
  - Dynamically allocate space for each element as needed
  - Include a pointer to the next item
- Linked list
  - Each node of the list contains
    - the data item (an object pointer in our ADT)
    - a pointer to the next node

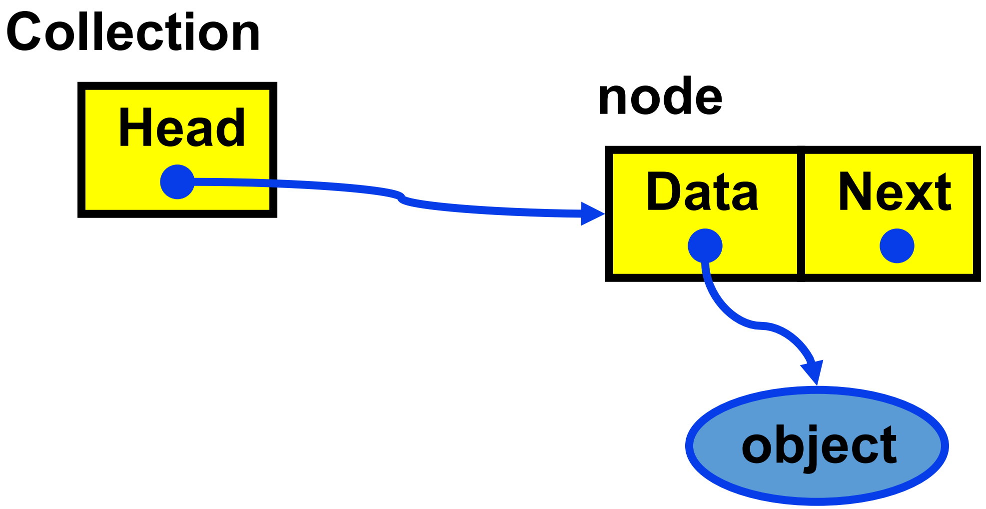


Linked Lists

- Collection structure has a pointer to the list head
  - Initially NULL

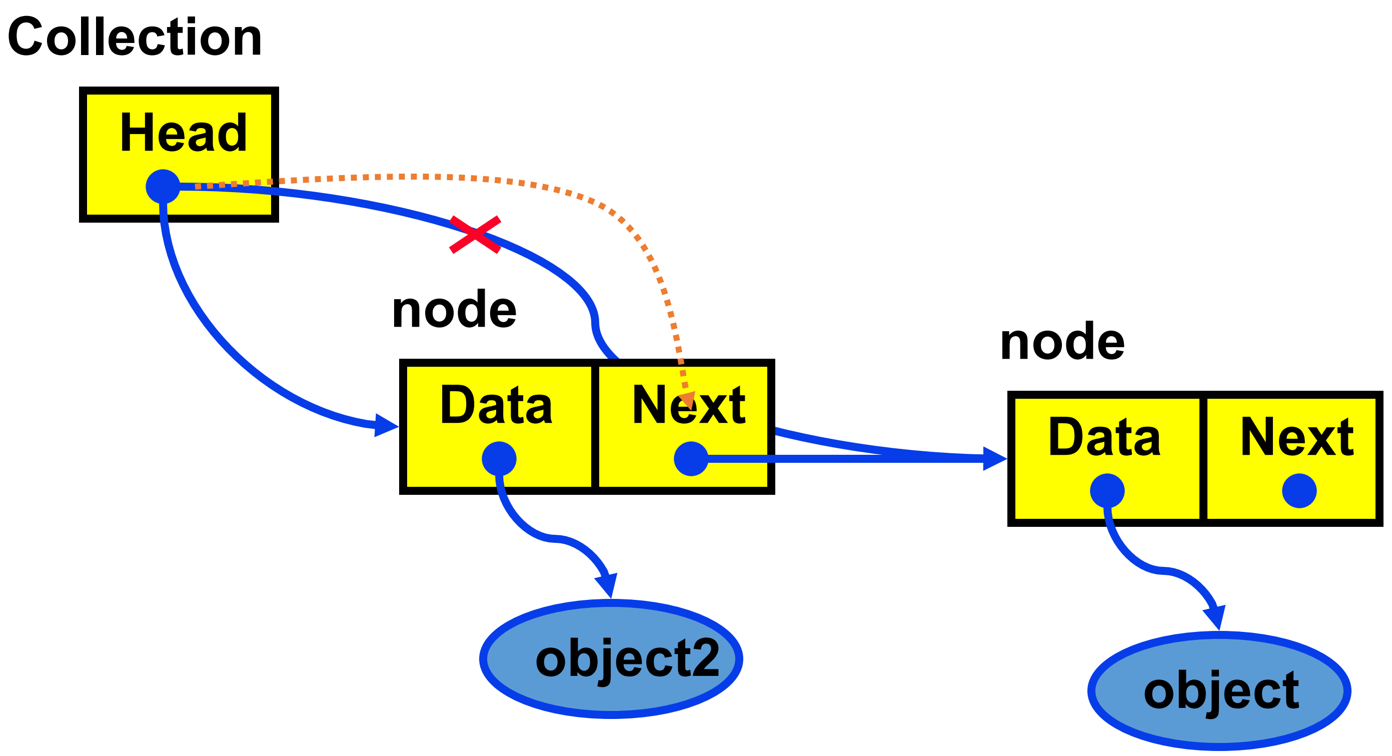
Linked Lists

- Collection structure has a pointer to the list head
  - Initially NULL
- Add first item
  - Allocate space for node
  - Set its data pointer to object
  - Set Next to NULL
  - Set Head to point to new node



Linked Lists

- Add second item
  - Allocate space for node
  - Set its data pointer to object
  - Set Next to current Head
  - Set Head to point to new node



Linked Lists C/C++

```
struct t_node {
    void *item;
    struct t_node *next;
    } node;
typedef struct t_node *Node;
struct collection {
    Node head;
    ……
    };
int AddToCollection( Collection c, void *item ) {
    Node new = malloc( sizeof( struct t_node ) );
    new->item = item;
    new->next = c->head;
    c->head = new;
    return TRUE;
    }
```

Linked Lists - C/C++

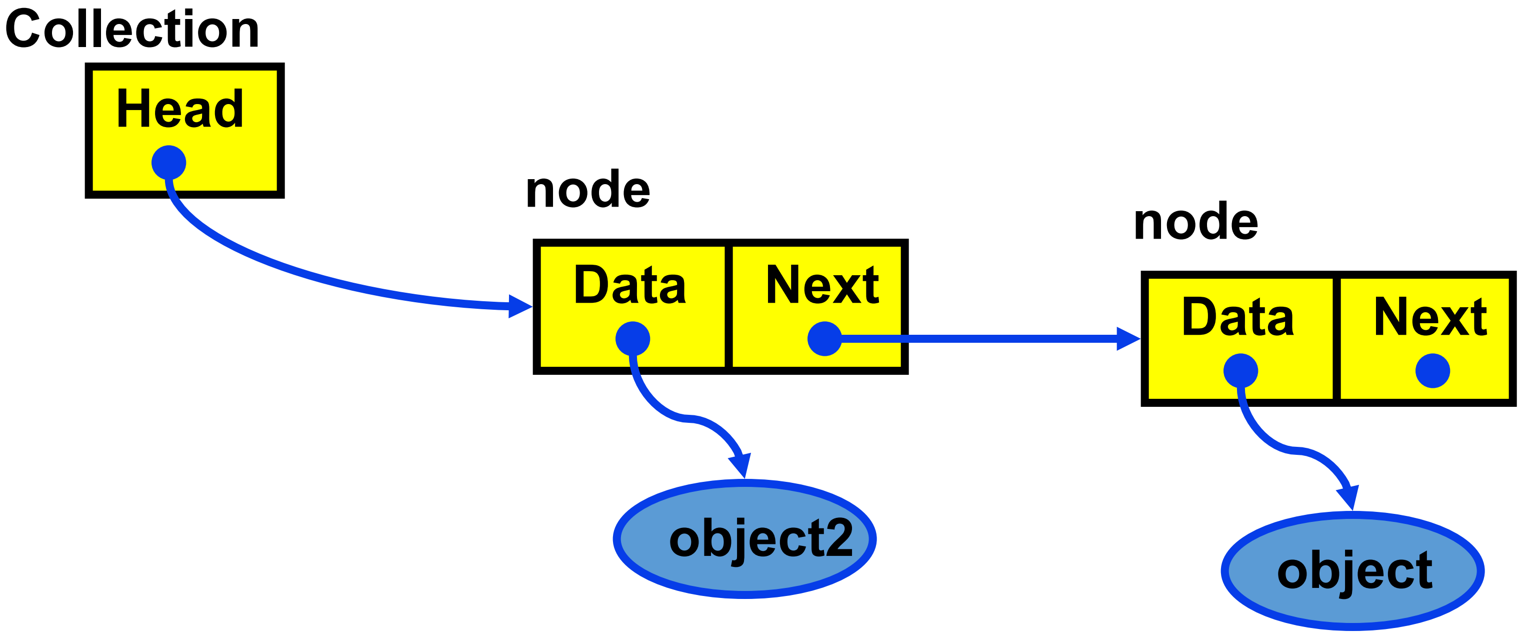
```
struct t_node {
    void *item;
    struct t_node *next;
    } node;
typedef struct t_node *Node;
struct collection {
    Node head;
    ……
    };
int AddToCollection( Collection c, void *item ) {
    Node new = malloc( sizeof( struct t_node ) );
    new->item = item;
    new->next = c->head;
    c->head = new;
    return TRUE;
    }
```

Recursive type definition - C allows it!

Error checking, asserts omitted for clarity!

Linked Lists

- Insertion/Deletion
  - Constant - independent of n
- Search time
  - Worst case - n

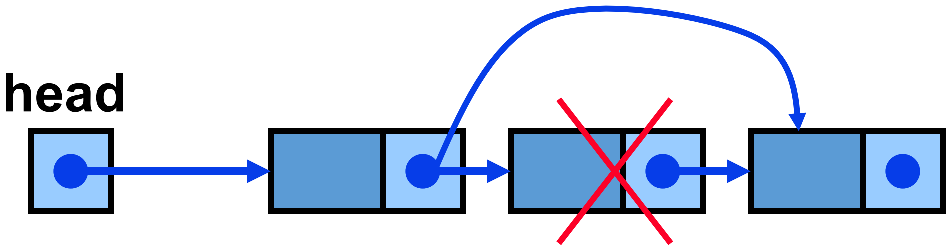


Linked Lists – C/C++

```
void *FindinCollection( Collection c, void *key ) {
    Node n = c->head;
    while ( n != NULL ) {
     if ( KeyCmp( ItemKey( n->item ), key ) == 0 ) {
          return n->item;
       n = n->next;
       }
    return NULL;
    }
```

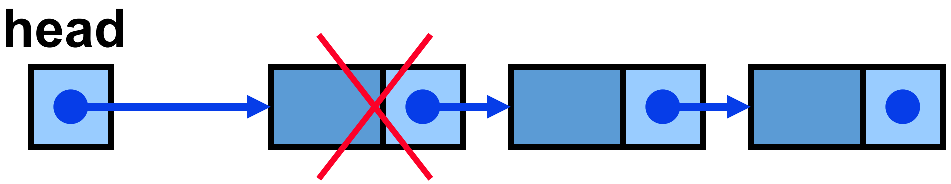
Linked Lists - Delete implementation

```
void *DeleteFromCollection( Collection c, void *key ) {
    Node n, prev;
    n = prev = c->head;
    while ( n != NULL ) {
      if ( KeyCmp( ItemKey( n->item ), key ) == 0 ) {
            prev->next = n->next;
            return n;
            }
      prev = n;
      n = n->next;
      }
    return NULL;
    }
```



Linked Lists - Delete implementation

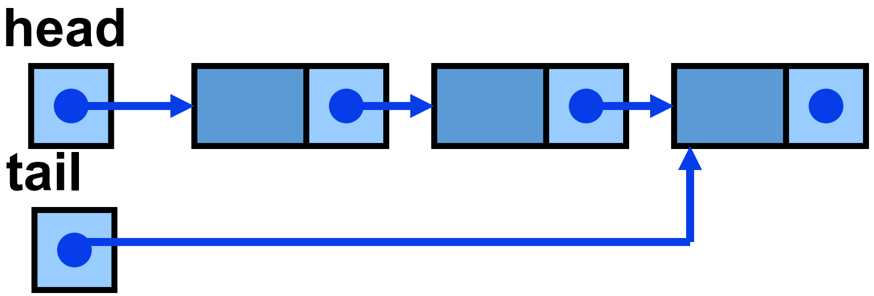
```
void *DeleteFromCollection( Collection c, void *key ) {
    Node n, prev;
    n = prev = c->head;
    while ( n != NULL ) {
      if ( KeyCmp( ItemKey( n->item ), key ) == 0 ) {
            prev->next = n->next;
            return n;
            }
      prev = n;
      n = n->next;
      }
    return NULL;
    }
```



Minor addition needed to allow for deleting this one! An exercise!

Linked Lists - LIFO and FIFO

- Simplest implementation
  - Add to head
  - ➧ Last-In-First-Out (LIFO) semantics
- Modifications
  - First-In-First-Out (FIFO)
  - Keep a tail pointer



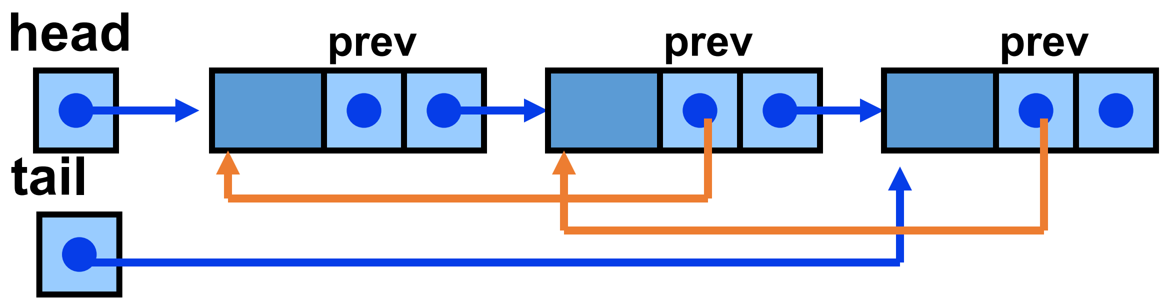

```
struct t_node {
     void *item;
     struct t_node *next;
     } node;
typedef struct t_node *Node;
struct collection {
     Node head, tail;
     };
```

tail is set in the `AddToCollection` method if `head == NULL`

Linked Lists - Doubly linked

- Doubly linked lists
  - Can be scanned in both directions

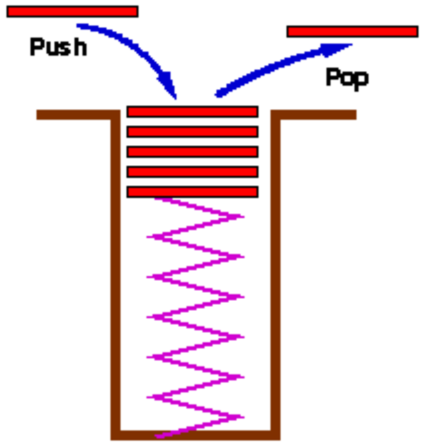
```
struct t_node {
    void *item;
    struct t_node *prev,
                  *next;
    } node;

typedef struct t_node *Node;
struct collection {
    Node head, tail;
    };
```



Stacks

- Stacks are a special form of collection with LIFO semantics
- Two methods
    - int push(Stack s, void *item);
      - add item to the top of the stack
    - void *pop(Stack s);
      - remove an item from the top of the stack
- Like a plate stacker
- Other methods

```
int IsEmpty( Stack s );
/* Return TRUE if empty */
void *Top( Stack s );
/* Return the item at the top,
    without deleting it */
```



Stacks - Implementation

- Arrays
  - Provide a stack capacity to the constructor
  - Flexibility limited *but* matches many real uses
    - Capacity limited by some constraint
      - Memory in your computer
      - Size of the plate stacker, etc
- push, pop methods
  - Variants of AddToC…, DeleteFromC…
- Linked list also possible

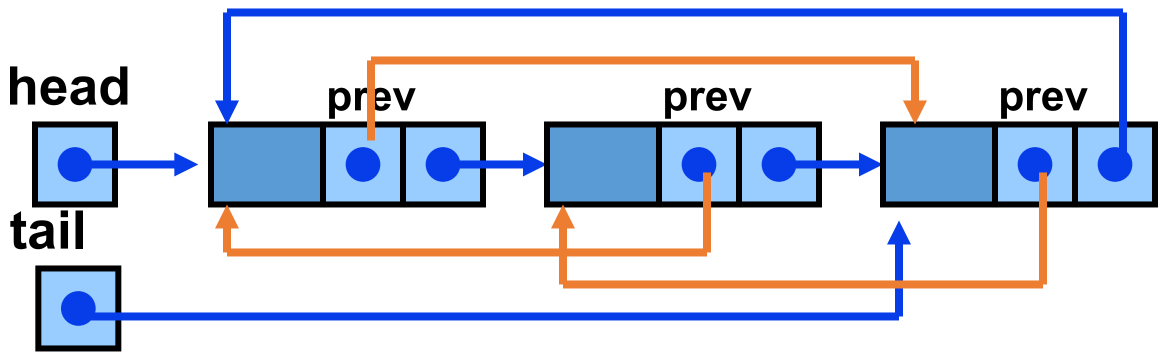
Stacks - Relevance

- Stacks appear in computer programs
  - Key to call / return in functions & procedures
  - Stack frame allows recursive calls
  - Call: push stack frame
  - Return: pop stack frame
- Stack frame
  - Function arguments
  - Return address
  - Local variables

Stacks - Implementation

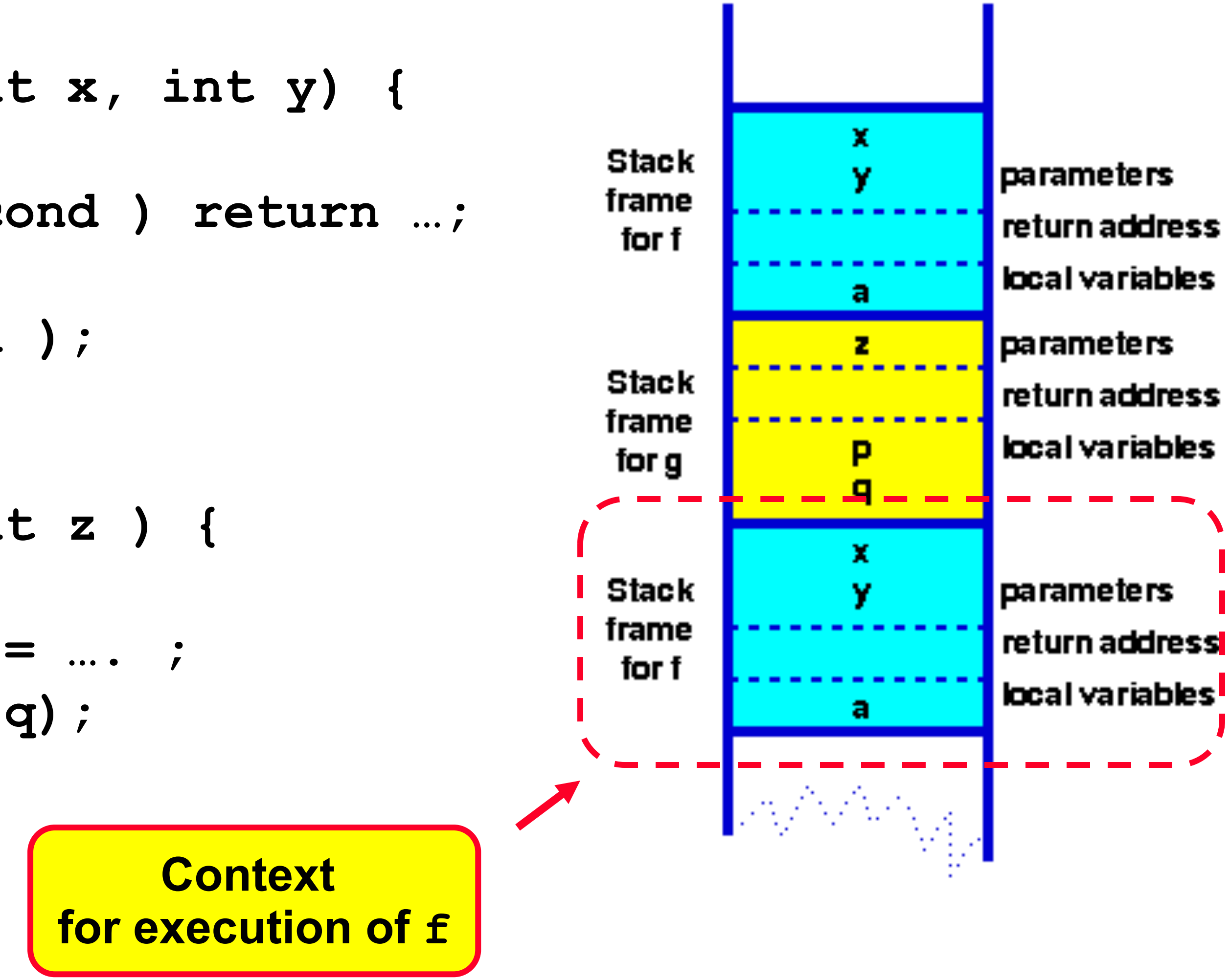
- Arrays common
  - Provide a stack capacity to the constructor
  - Flexibility limited but matches many real uses
    - Stack created with limited capacity

```
struct t_node {
    void *item;
    struct t_node *prev,
                  *next;
    } node;

typedef struct t_node *Node;
struct collection {
    Node head, tail;
    };
```



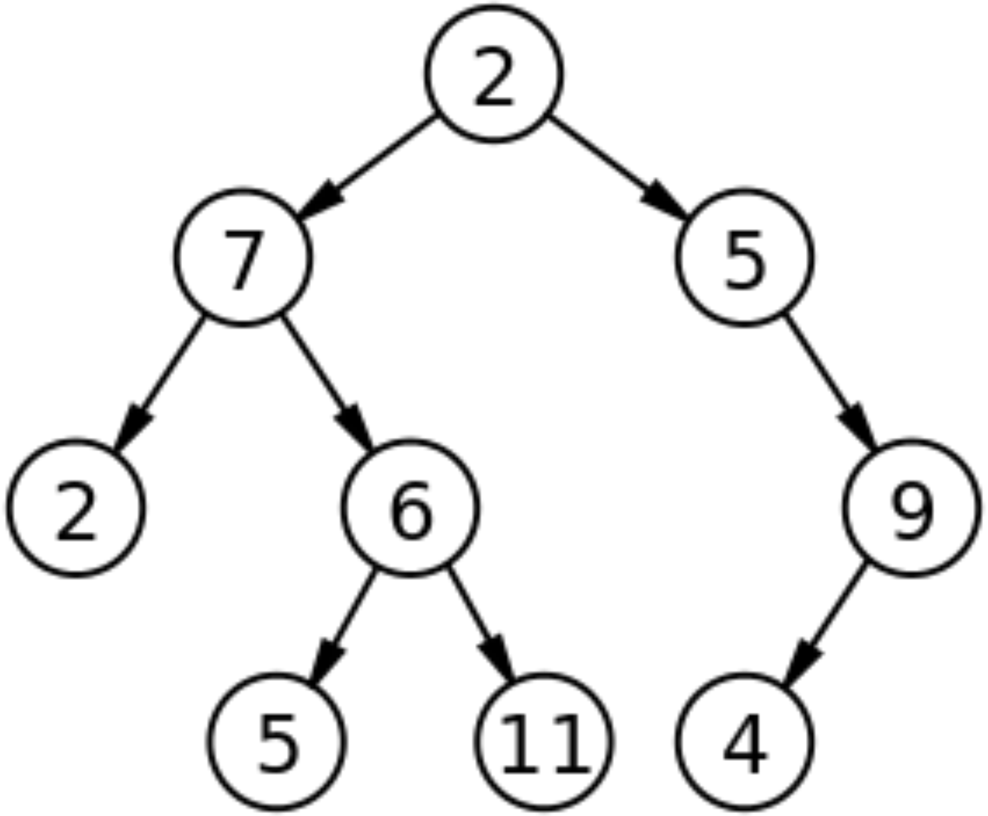
Stack Frames - Functions in HLL

```
function f( int x, int y) {
   int a;
   if ( term_cond ) return …;
   a = ….;
   return g( a );
   }

function g( int z ) {
   int p, q;
   p = …. ; q = …. ;
   return f(p,q);
   }
```



Binary Trees

- Binary Tree
  - Consists of
    - Node
    - Left and Right sub-trees
    - Both sub-trees are binary trees



Trees - Implementation

```
struct t_node {
	void *item;
	struct t_node *left;
	struct t_node *right;
	};

typedef struct t_node *Node;

struct t_collection {
	Node root;
	……
	};
```

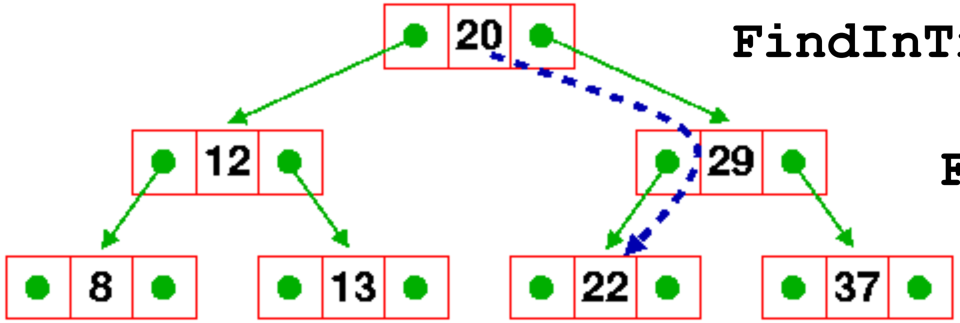
Trees - Implementation

```
extern int KeyCmp( void *a, void *b );
/* Returns -1, 0, 1 for a < b, a == b, a > b */

void *FindInTree( Node t, void *key ) {
   if ( t == (Node)0 ) return NULL;
   switch( KeyCmp( key, ItemKey(t->item) ) ) {
      case -1 : return FindInTree( t->left, key );
      case 0:   return t->item;
      case +1 : return FindInTree( t->right, key );
      }
   }

void *FindInCollection( collection c, void *key ) {
   return FindInTree( c->root, key );
   }
```

Trees - Implementation

- Find
  - key = 22;
    if (FindInCollection(c , &key)) ….

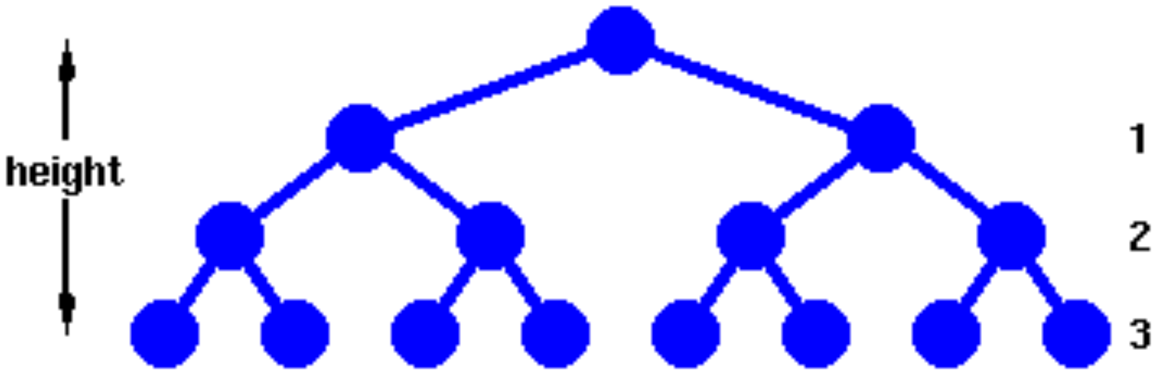
```
n = c->root;
FindInTree( n, &key );
```




```
FindInTree(n->right,&key );
FindInTree(n->left,&key );
return n->item;
```

Trees - Performance

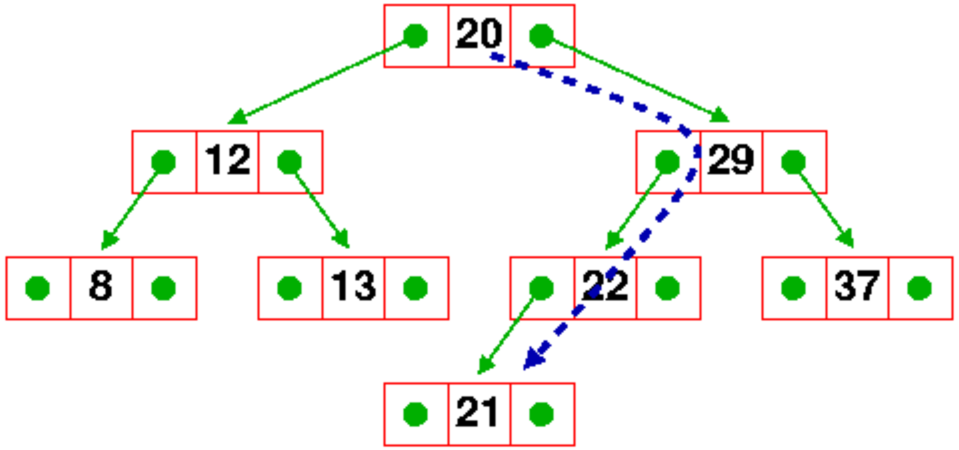
- Find
  - Complete Tree



  - Height, h
    - Nodes traversed in a path from the root to a leaf
  - Number of nodes, h
    - $n = 1 + 2^1 + 2^2 + \ldots + 2^h = 2^{h+1} - 1$
    - $h = \text{floor}(\log_2 n)$

Trees - Addition

- Add 21 to the tree



- We need at most $h+1$ comparisons
- Create a new node (constant time)

$\therefore$ add takes $c_1(h+1)+c_2$ or $c \log n$

- So addition to a tree takes time proportional to $\log n$ also

Trees - Addition - implementation

```
static void AddToTree( Node *t, Node new ) {
   Node base = *t;
   /* If it's a null tree, just add it here */
   if ( base == NULL ) {
      *t = new; return; }
   else
      if( KeyLess(ItemKey(new->item),ItemKey(base->item)) )
         AddToTree( &(base->left), new );
      else
         AddToTree( &(base->right), new );
   }

void AddToCollection( collection c, void *item ) {
        Node new, node_p;
        new = (Node)malloc(sizeof(struct t_node));
        /* Attach the item to the node */
        new->item = item;
        new->left = new->right = (Node)0;
        AddToTree( &(c->node), new );
        }
```

Trees - Addition

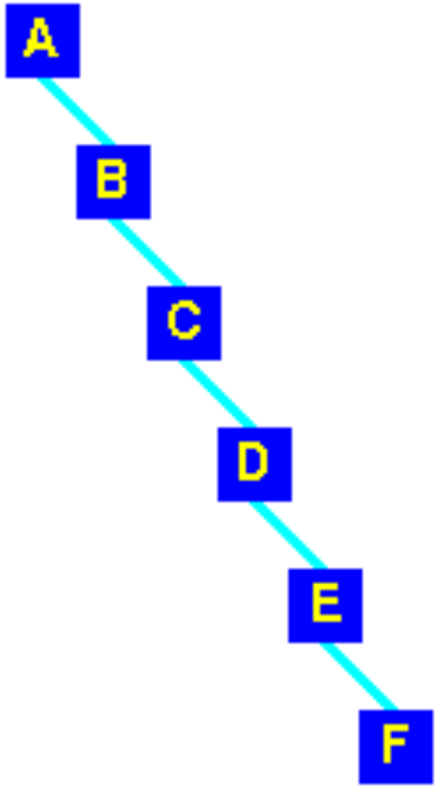
- Find $c \log n$
- Add $c \log n$
- Delete $c \log n$

- Usually efficient in every respect!

- *But there's a catch ……….. Balance!!!*

Trees - Addition

- Take this list of characters and form a tree
  A B C D E F



- In this case
  ? Find
  ? Add
  ? Delete

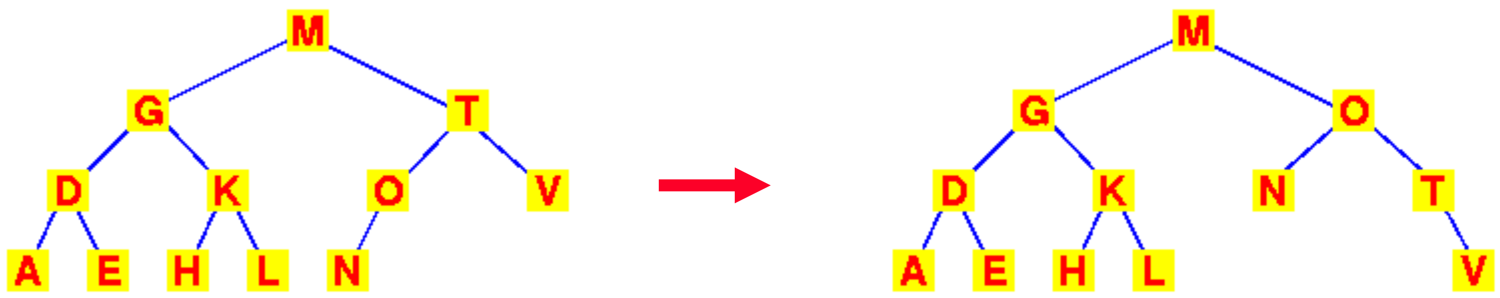
Searching - Re-visited

- Binary tree *O(log n) if it stays balanced*
  - Simple binary tree good for static collections
  - Low (preferably zero) frequency of insertions/deletions

but my collection keeps changing!
  - It's dynamic
  - Need to keep the tree balanced
- First, examine some basic tree operations
  - Useful in several ways!

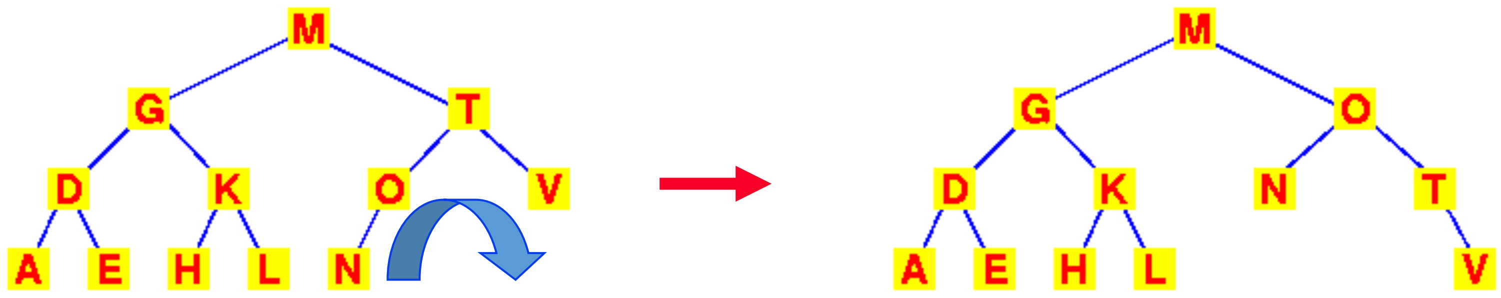
Trees - Searching

- Binary search tree
  - Preserving the order
  - Observe that this transformation preserves the search tree



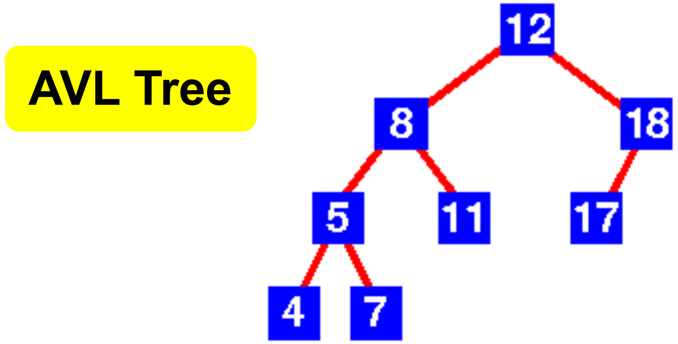
Trees - Searching

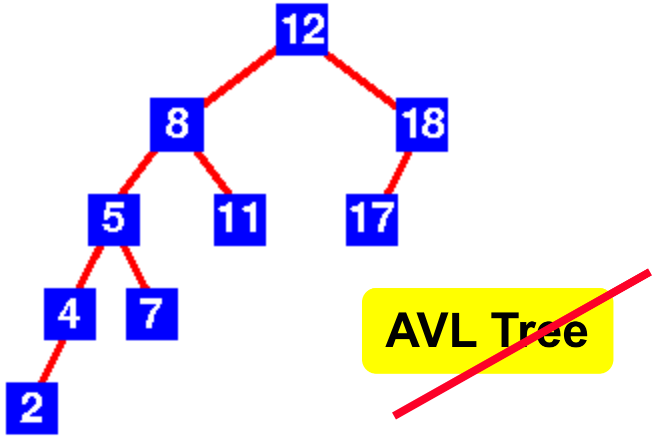
- Binary search tree
  - Preserving the order
  - Observe that this transformation preserves the search tree
- We've performed a rotation of the sub-tree about the T and O nodes



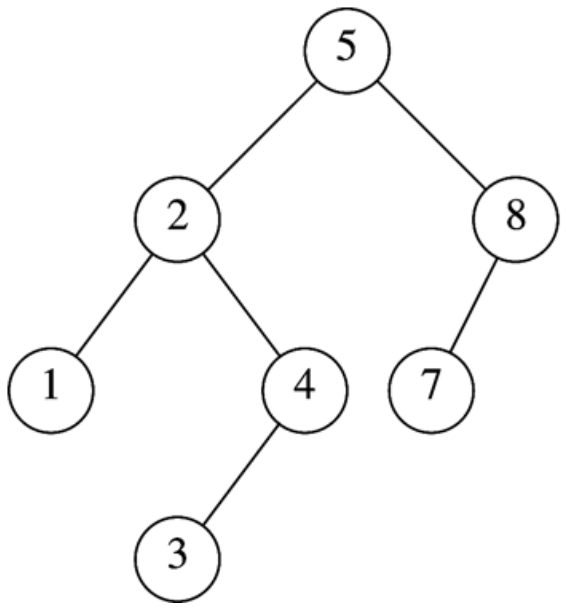
AVL and other balanced trees

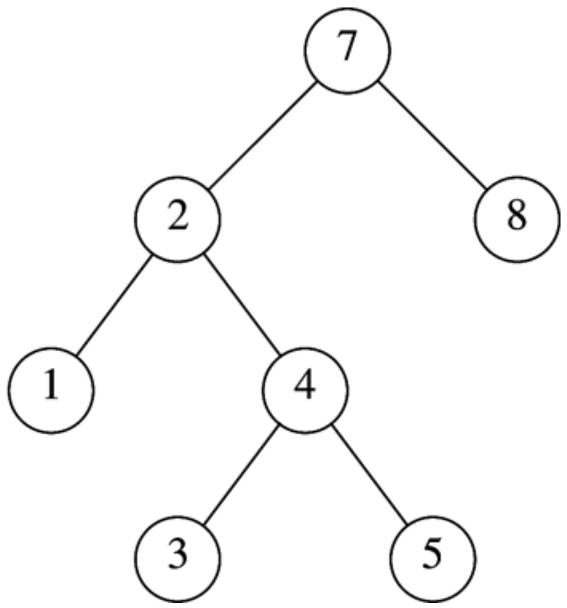
- AVL Trees
  - First balanced tree algorithm
  - Discoverers: Adelson-Velskii and Landis
- Properties
  - Binary tree
  - Height of left and right-subtrees differ by at most 1
  - Subtrees are AVL trees





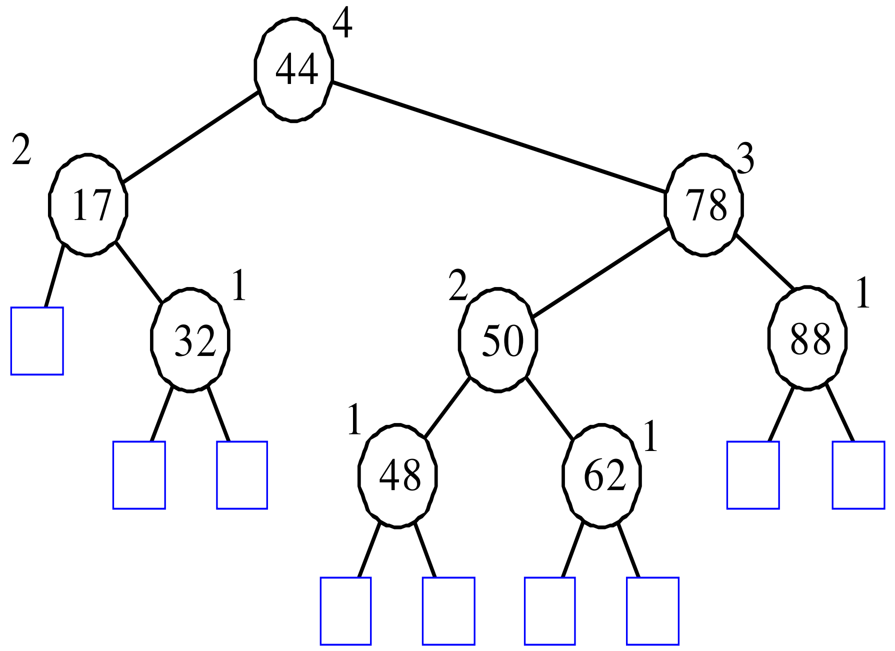
Which is an AVL Tree?





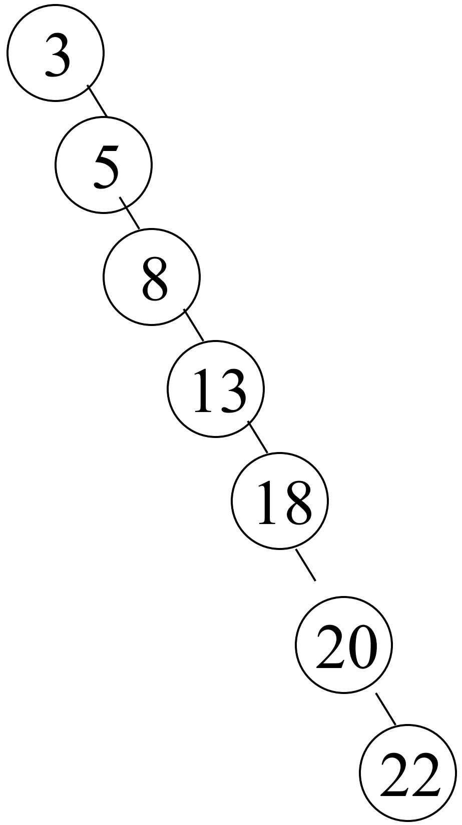
AVL (Adelson-Velskii and Landis) Trees

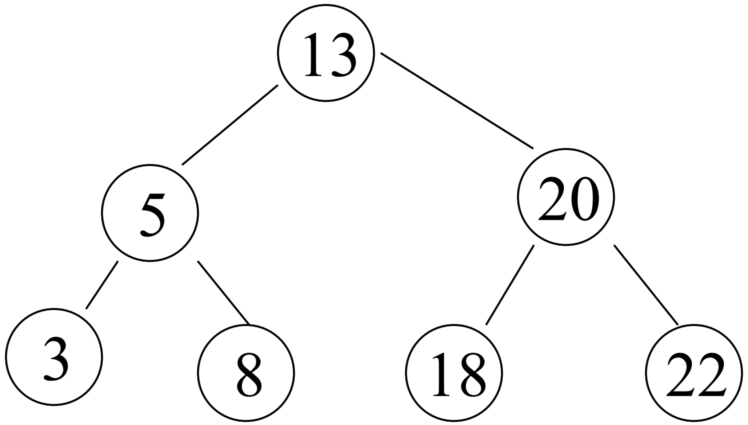
An AVL Tree is a ***binary search tree*** such that for every internal node v of T, the *heights of the children of v can differ by at most 1.*



Motivation

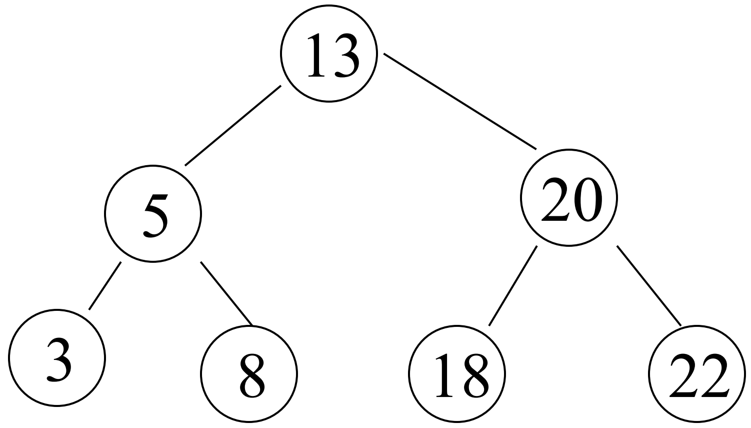
When building a binary search tree, what type of trees would we like? Example: 3, 5, 8, 20, 18, 13, 22

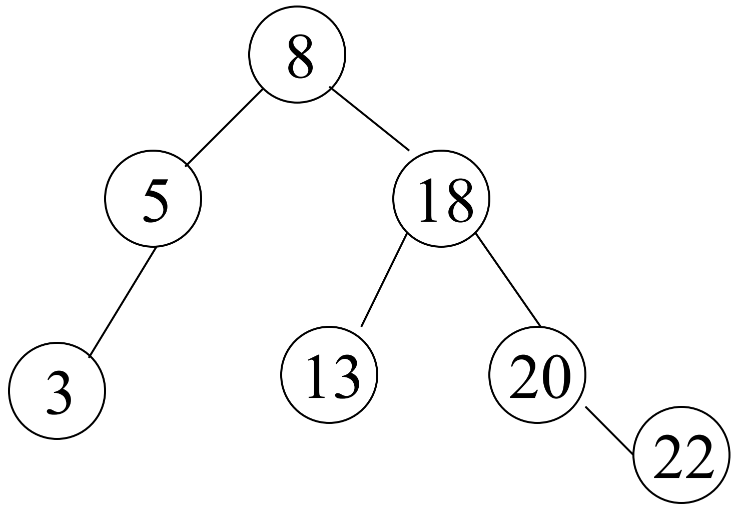




Motivation

- Complete binary tree is hard to build when we allow dynamic insert and remove.
  - We want a tree that has the following properties
    - Tree height = O(log(N))
    - allows dynamic insert and remove with O(log(N)) time complexity.
  - The AVL tree is one of this kind of trees.





AVL (Adelson-Velskii and Landis) Trees

- AVL tree is a binary search tree with balance condition
  - To ensure depth of the tree is O(log(N))
  - And consequently, search/insert/remove complexity bound O(log(N))
- Balance condition
  - For every node in the tree, height of left and right subtree can differ by at most 1

Height of an AVL tree

- Theorem: The ***height*** of an AVL tree storing n keys is O(log n).
- Proof:
    - Let us bound **n(h),** the minimum number of internal nodes of an AVL tree of height h.
    - We easily see that $n(0) = 1$ and $n(1) = 2$
    - For $h > 2$, an AVL tree of height h contains the root node, one AVL subtree of height h-1 and another of height h-2 (at worst).
    - That is, $n(h) >= 1 + n(h-1) + n(h-2)$
    - Knowing $n(h-1) > n(h-2)$, we get $n(h) > 2n(h-2)$. So

      $n(h) > 2n(h-2)$, $n(h) > 4n(h-4)$, $n(h) > 8n(n-6)$, … (by induction),

      $n(h) > 2^i n(h-2i)$
    - Solving the base case we get: $n(h) > 2^{h/2-1}$
    - Taking logarithms: $h < 2\log n(h) +2$
    - Since $n>=n(h)$, $h < 2\log(n)+2$ and the height of an AVL tree is O(log n)

AVL Trees - Data Structures

```
typedef enum { LeftHeavy, Balanced, RightHeavy }
                        BalanceFactor;
struct AVL_node {
    BalanceFactor bf;
    void *item;
    struct AVL_node *left, *right;
    }
```

- Insertion
  - Insert a new node (as any binary tree)
  - Work up the tree re-balancing as necessary to restore the AVL property

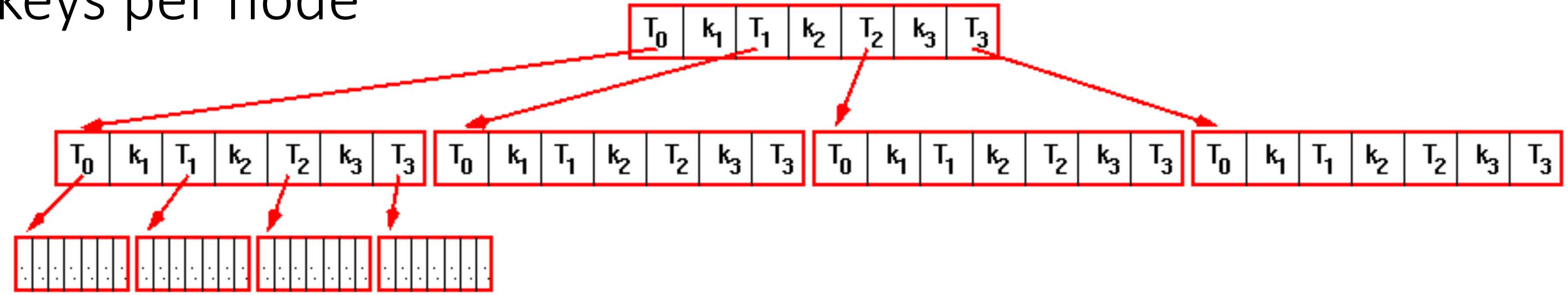
m-way trees (Multiway Trees)

- A multiway tree is a tree that can have more than two children. A multiway tree of order m (or an m-way tree) is one in which a tree can have m children.
- But you have to search through the *m* keys in each node!
- Reduces your gain from having fewer levels.

m-way trees

- Only two children per node?
- Reduce the depth of the tree to $O(\log_m n)$ with *m*-way trees

- *m* children, *m*-1 keys per node



- $m = 10$: 10^6 keys in 6 levels *vs* 20 for a binary tree

B-trees

- All leaves are on the same level
- All nodes except for the root and the leaves have
  - at least *m/2* children
  - at most *m* children
- B+ trees
  - All the keys in the nodes are dummies
  - Only the keys in the leaves point to “real” data
  - Linking the leaves
    - Ability to scan the collection *in order* without passing through the higher nodes

Motivation for B-Trees

- Index structures for large datasets cannot be stored in main memory
- Storing it on disk requires different approach to efficiency

- Assuming that a disk spins at 3600 RPM, one revolution occurs in 1/60 of a second, or 16.7ms
- Crudely speaking, one disk access takes about the same time as 200,000 instructions

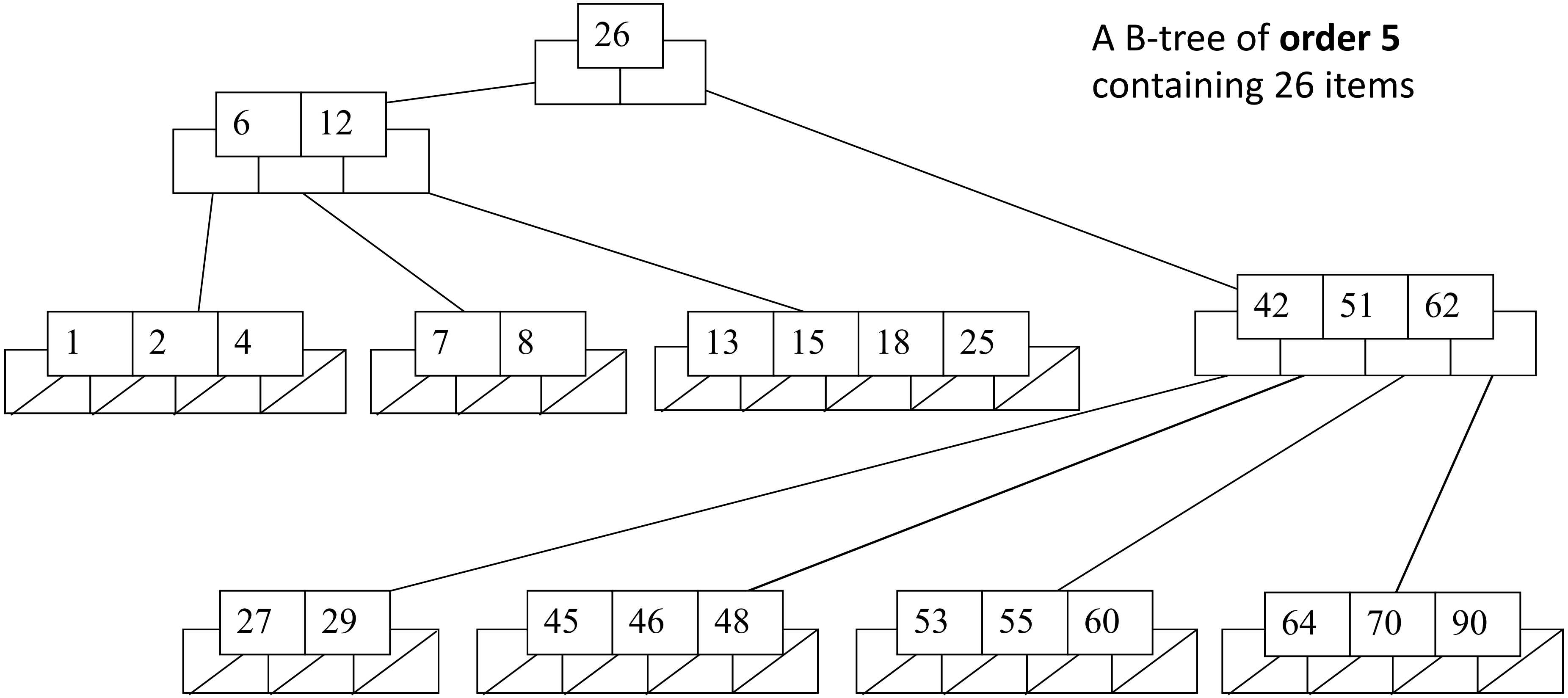
Motivation B-trees

- Assume that we use an AVL tree to store about 20 million records
- We end up with a **very** deep binary tree with lots of different disk accesses; $\log_2 20{,}000{,}000$ is about 24, so this takes about 0.2 seconds
- We know we can't improve on the log *n* lower bound on search for a binary tree
- But, the solution is to use more branches and thus reduce the height of the tree!
  - As branching increases, depth decreases

Definition of B-Tree

- Definition assumes external nodes (extended m-way search tree).
- B-tree of order m.
  - m-way search tree.
  - Not empty => root has at least 2 children.
  - Remaining internal nodes (if any) have at least ceil(m/2) children.
  - External (or failure) nodes on same level.

An example B-Tree

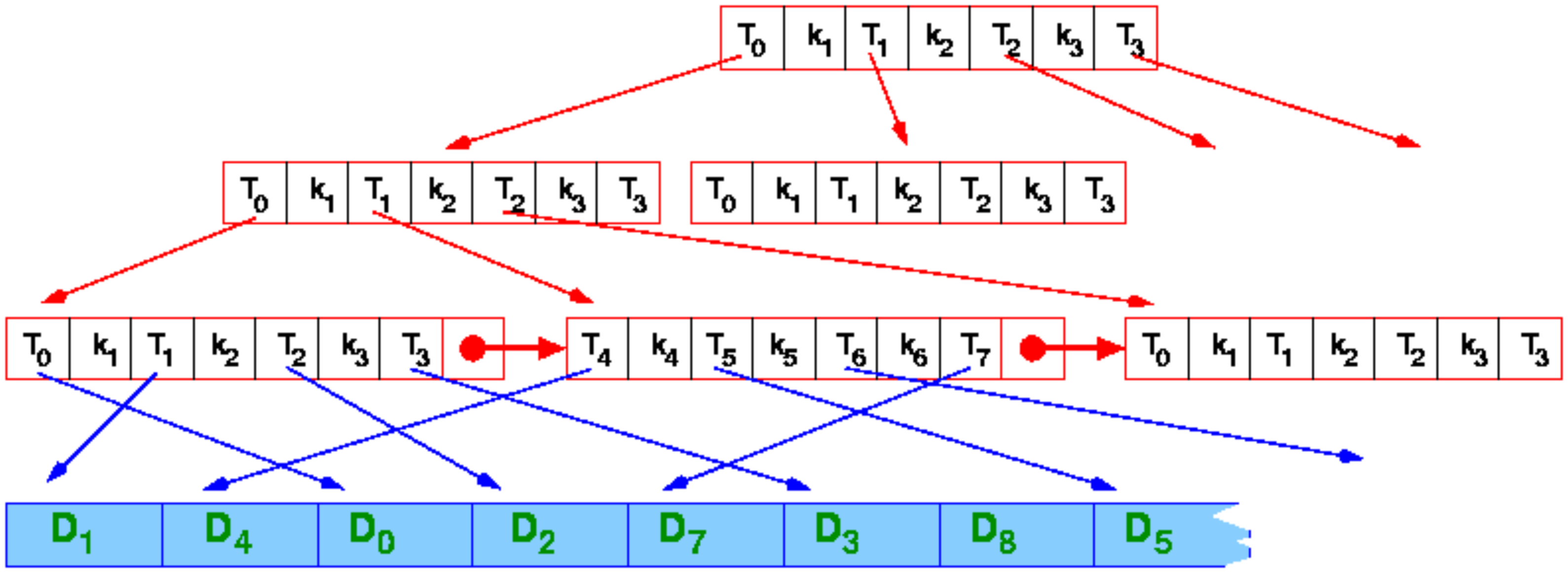


A B-tree of **order 5** containing 26 items

Note that all the leaves are at the same level

B+-trees

- B+ trees
  - All the keys in the nodes are dummies
  - Only the keys in the leaves point to “real” data
  - Data records kept in a separate area



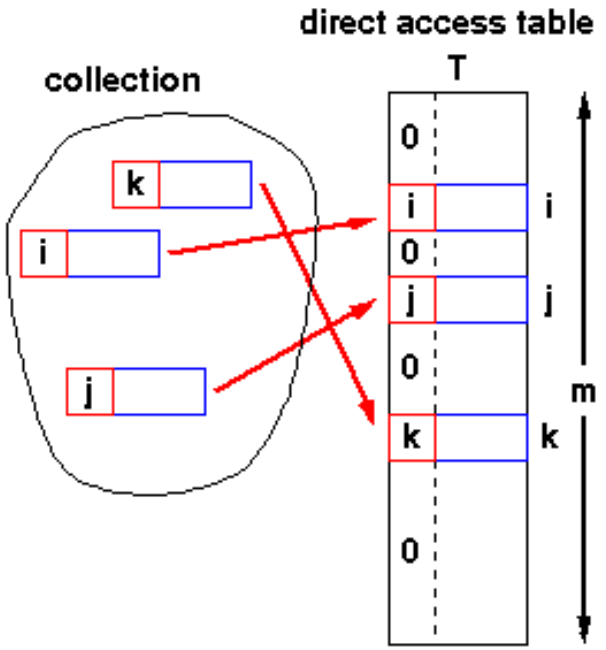
Hash Tables

- All search structures so far
  - Relied on a comparison operation
  - Performance $O(n)$ or $O(\log n)$
- Assume I have a function
  - *f (key)* ® *integer*

    ie one that maps a key to an integer
- What performance might I expect now?

Hash Tables - Structure

- Simplest case:
  - Assume items have integer keys in the range
  - Use the value of the key itself
    to select a slot in a
    direct access table
    in which to store the item
  - To search for an item with key, *k*,
    just look in slot *k*
    - If there's an item there,
      you've found it
    - If the tag is 0, it's missing.
  - Constant time, *O(1)*



Hash Tables - Constraints

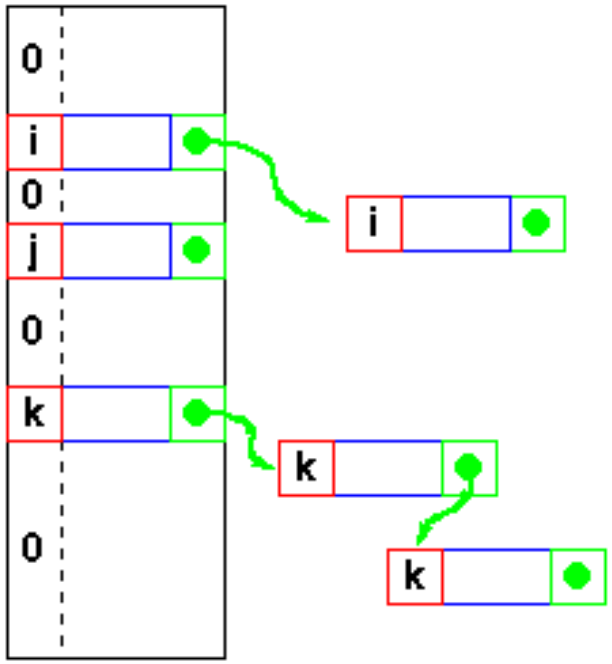
- Constraints
  - Keys must be unique
  - Keys must lie in a small range
  - For storage efficiency,
    keys must be dense in the range
  - If they're sparse (lots of gaps between values),
    a lot of space is used to obtain speed
    - Space for speed trade-off

Hash Tables - Relaxing the constraints

- Keys must be unique
  - Construct a linked list of duplicates "attached" to each slot
  - If a search can be satisfied by *any* item with key, *k*, performance is still $O(1)$

    but
  - If the item has some other distinguishing feature which must be matched, we get $O(n^{max})$

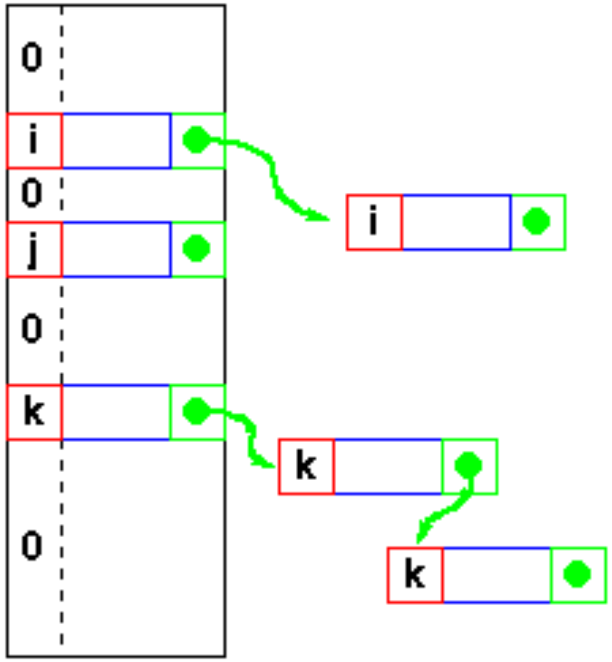
    where n^{max} is the largest number of duplicates - or length of the longest chain



Hash Tables - Relaxing the constraints

- Keys are integers
  - Need a hash function
    h(key) ® *integer*

    ie one that maps a key to an integer
  - Applying this function to the key produces an address
  - If *h* maps each key to a *unique integer* in the range 0 .. *m*-1 then search is *O(1)*



Hash Tables - Hash functions

- Example - using an *n*-character key

```
int hash( char *s, int n ) {
    int sum = 0;
    while( n-- ) sum = sum + *s++;
    return sum % 256;
    }
```

  returns a value in 0 .. 255
- xor function is also commonly used

  sum = sum ^ *s++;
- But any function that generates integers in 0..m-1 for some suitable (*not too large*) m will do

Hash Tables - Collisions

- Hash function
  - With this hash function

```
int hash( char *s, int n ) {
    int sum = 0;
    while( n-- ) sum = sum + *s++;
    return sum % 256;
    }
```

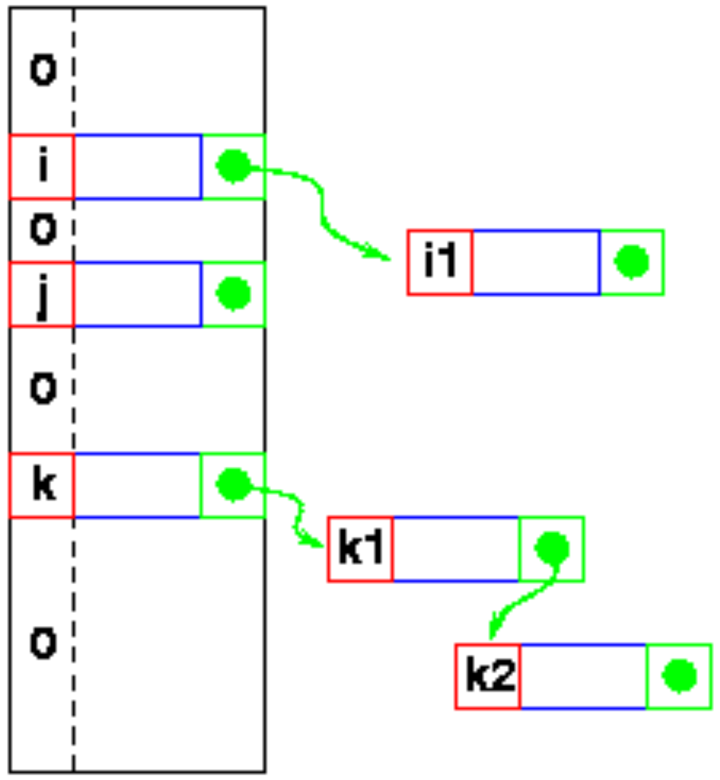
  - hash("AB", 2) and
    hash("BA", 2)
    return the same value!
  - This is called a collision
  - A variety of techniques are used for resolving collisions

Hash Tables - Collision handling

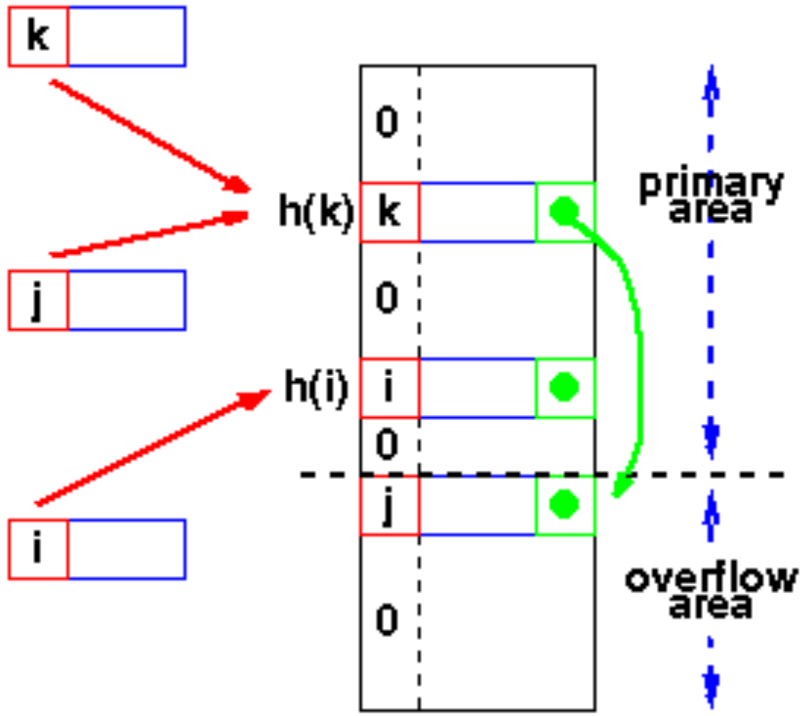
- Collisions
  - Occur when the hash function maps two different keys to the same address
  - The table must be able to recognise and resolve this
  - Recognise
    - Store the actual key with the item in the hash table
    - Compute the address
      - k = h(key)
    - Check for a hit
      - *if (* table[k].key == key *) then hit*
        else try next entry
  - Resolution
    - Variety of techniques

Hash Tables - Linked lists

- Collisions - Resolution
  - ❶Linked list attached
    to each primary table slot
    - h(i) == h(i1)
    - h(k) == h(k1) == h(k2)
  - Searching for i1
    - Calculate h(i1)
    - Item in table, i,
      doesn’t match
    - Follow linked list to i1
  - If NULL found,
    key isn’t in table



Hash Tables - Overflow area



Overflow area

- Linked list constructed in special area of table called overflow area

h(k) == h(j)

k stored first

Adding j

- Calculate h(j)
- Find k
- Get first slot in overflow area
- Put j in it
- k's pointer points to this slot

Searching - same as linked list

Hash Tables - Re-hashing

Use a second hash function

- Many variations
- General term: re-hashing

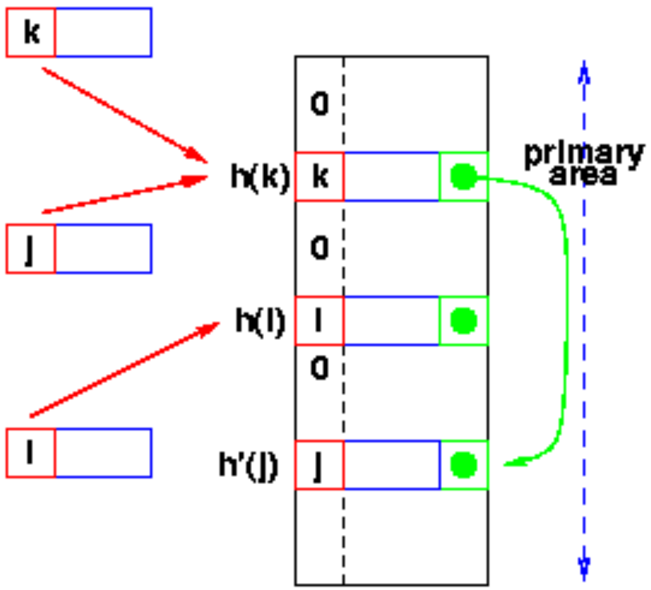
$h(k) == h(j)$

k stored first

Adding j

- Calculate $h(j)$
- Find k
- Repeat until we find an empty slot
  - Calculate $h'(j)$
- Put j in it

Searching - Use $h(x)$, then $h'(x)$



Hash Tables - Re-hash functions

The re-hash function

Many variations

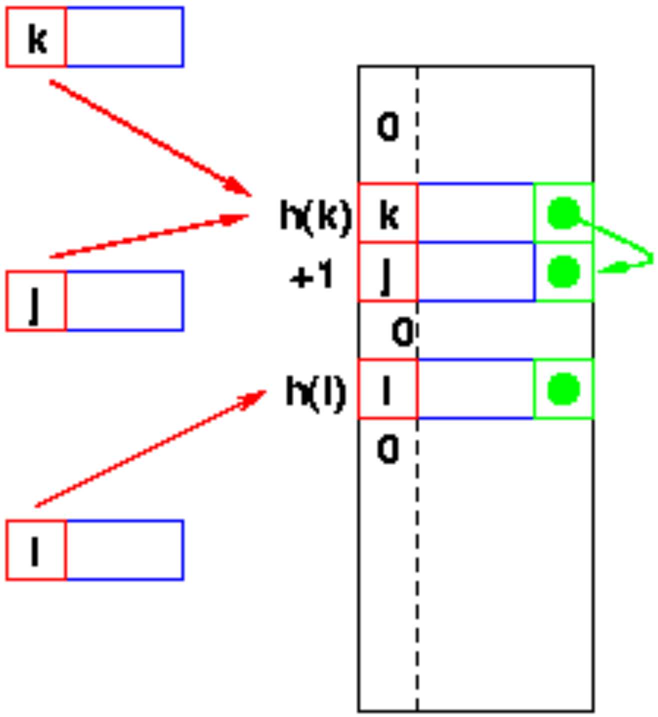
Linear probing

h'(x) is +1

Go to the next slot
until you find one empty

Can lead to bad clustering

Re-hash keys fill in gaps
between other keys and exacerbate
the collision problem



Hash Tables - Summary so far ...

- Potential *O(1)* search time
    - If a suitable function *h(key)* ® *integer* can be found
- Space for speed trade-off
    - "Full" hash tables don't work (more later!)
- Collisions
    - Inevitable
        - Hash function reduces amount of information in key
    - Various resolution strategies
        - Linked lists
        - Overflow areas
        - Re-hash functions
            - Linear probing h' is $+1$
            - Quadratic probing h' is $+ci^2$
            - Any other hash function!
                - or even sequence of functions!

Hash Tables - Choosing the Hash Function

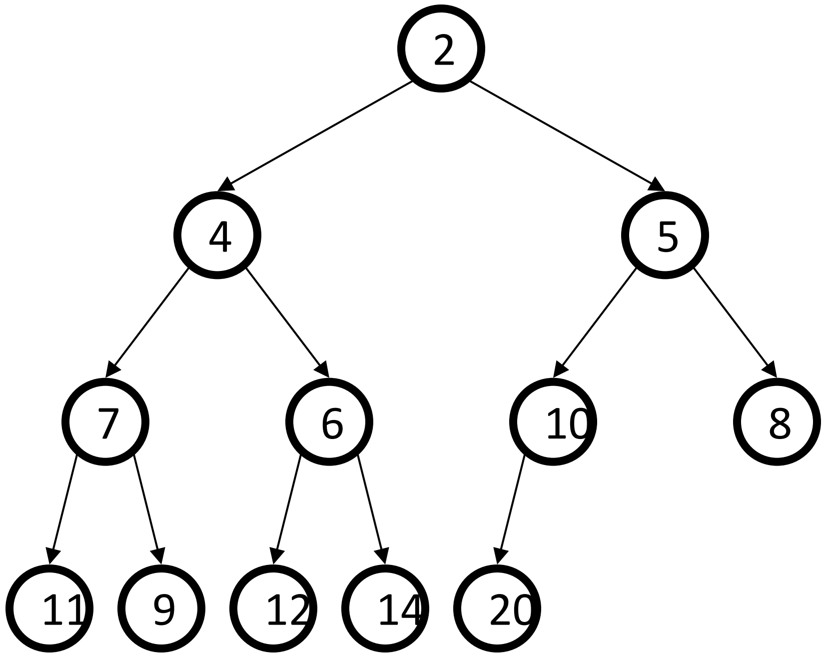
- "Almost any function will do"
  - But some functions are definitely better than others!
- Key criterion
  - Minimum number of collisions
    - Keeps chains short
    - Maintains $O(1)$ average

Collision Frequency

- Birthdays *or* the von Mises paradox
  - There are 365 days in a normal year
    - ➧ Birthdays on the same day unlikely?
  - How many people do I need
    before "it's an even bet"
    (*ie* the probability is > 50%)
    that two have the same birthday?
  - View
    - the days of the year as the slots in a hash table
    - the "birthday function" as mapping people to slots
  - Answering von Mises' question answers the question about the probability of collisions in a hash table

Binary Heap Priority Q Data Structure

- Heap-order property
  - parent's key is less than children's keys
  - result: minimum is always at the top
- Structure property
  - complete tree with fringe nodes packed to the left
  - result: depth is always O(log n); next open location always known



How do we find the minimum?

Heaps

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

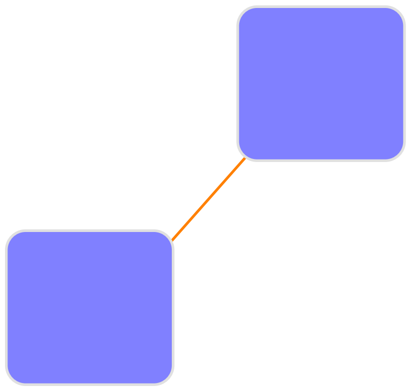
Heaps

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

When a complete binary tree is built, its first node must be the root.

Heaps

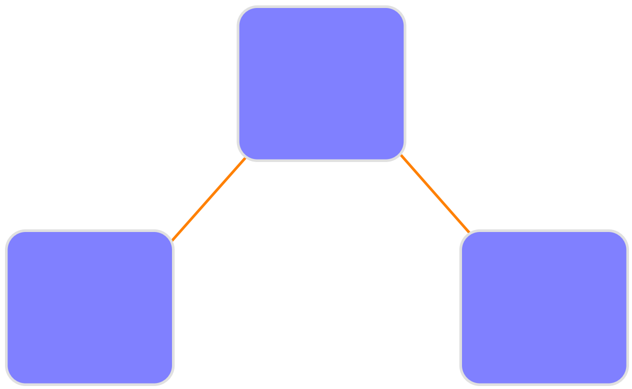
Complete binary tree.



The second node is always the left child of the root.

Heaps

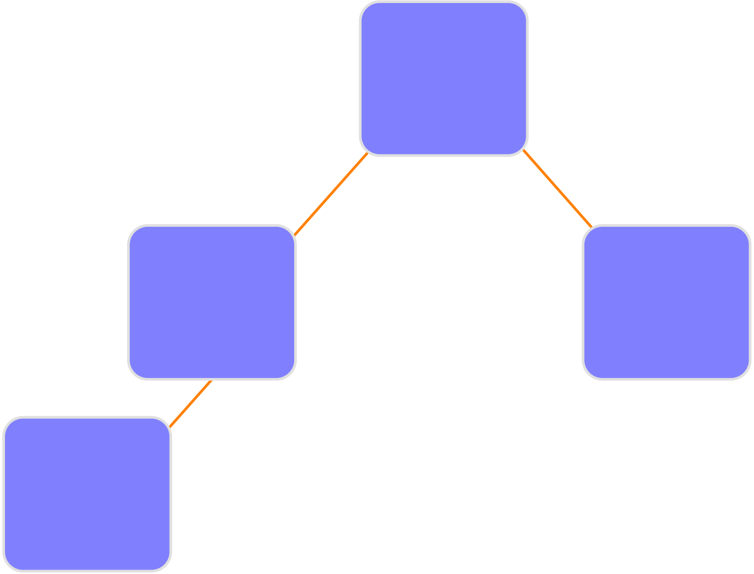
Complete binary tree.



The third node is always the right child of the root.

Heaps

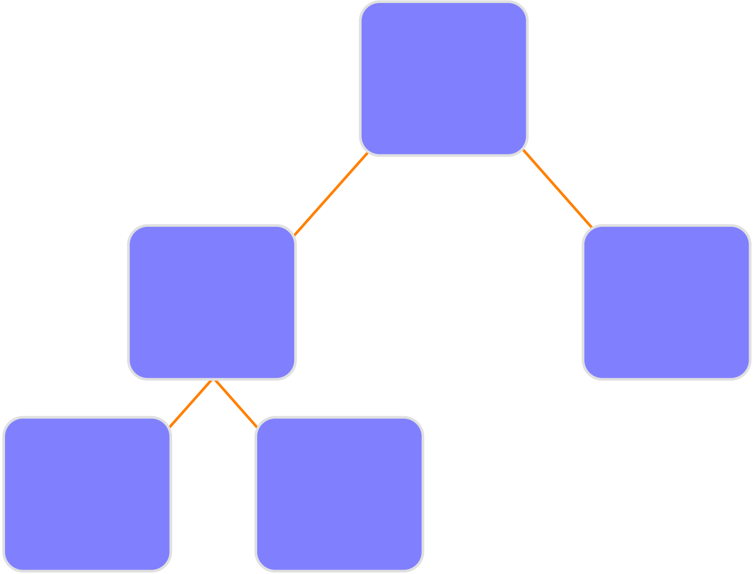
Complete binary tree.



The next nodes always fill the next level from left-to-right.

Heaps

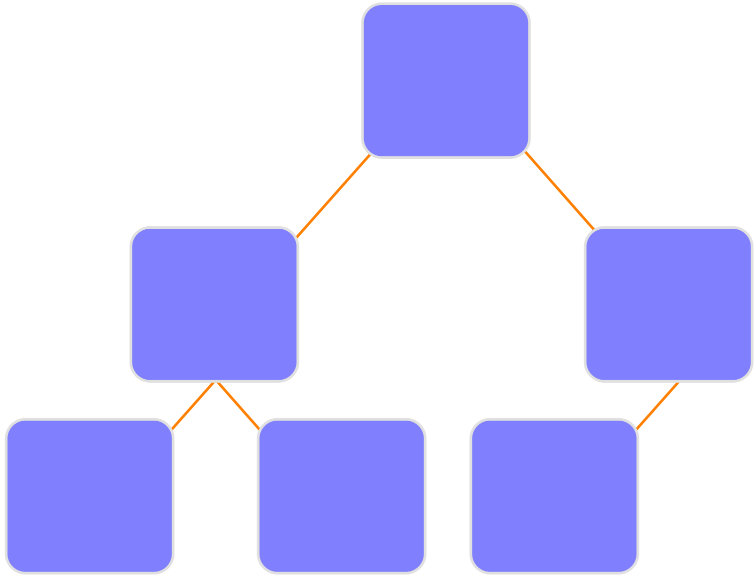
Complete binary tree.



The next nodes always fill the next level from left-to-right.

Heaps

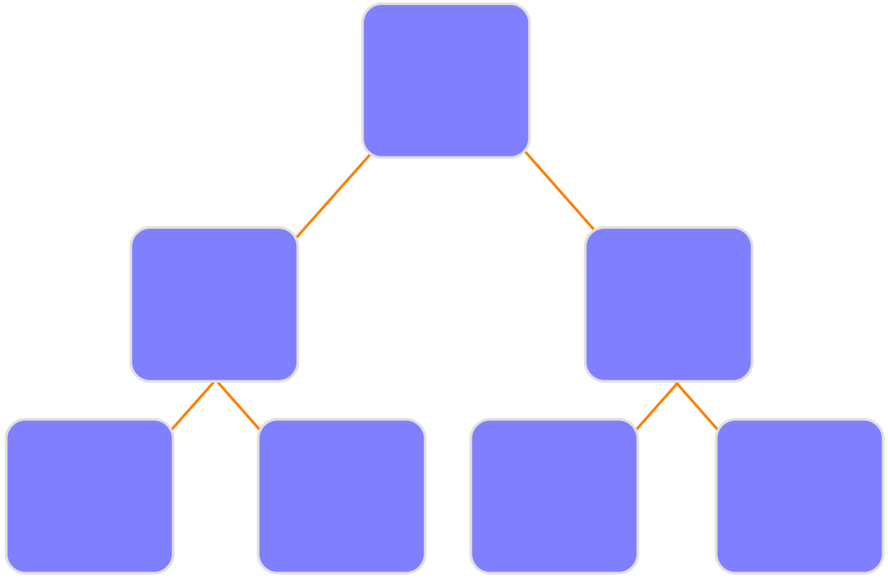
Complete binary tree.



The next nodes always fill the next level from left-to-right.

Heaps

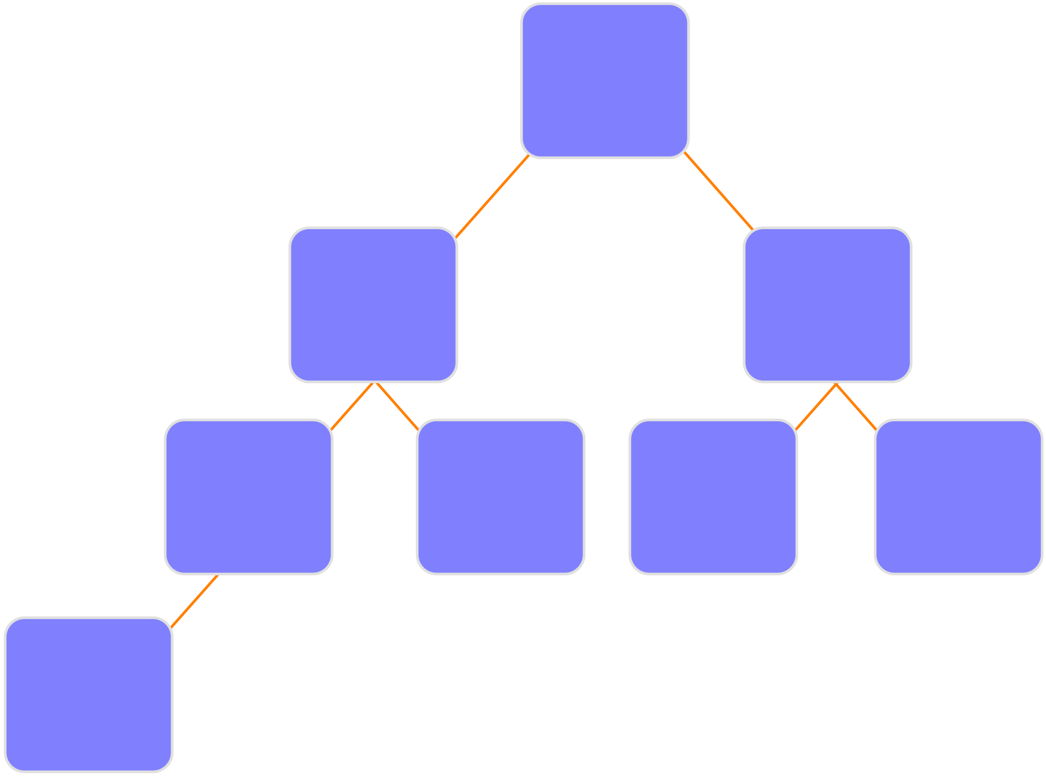
Complete binary tree.



The next nodes always fill the next level from left-to-right.

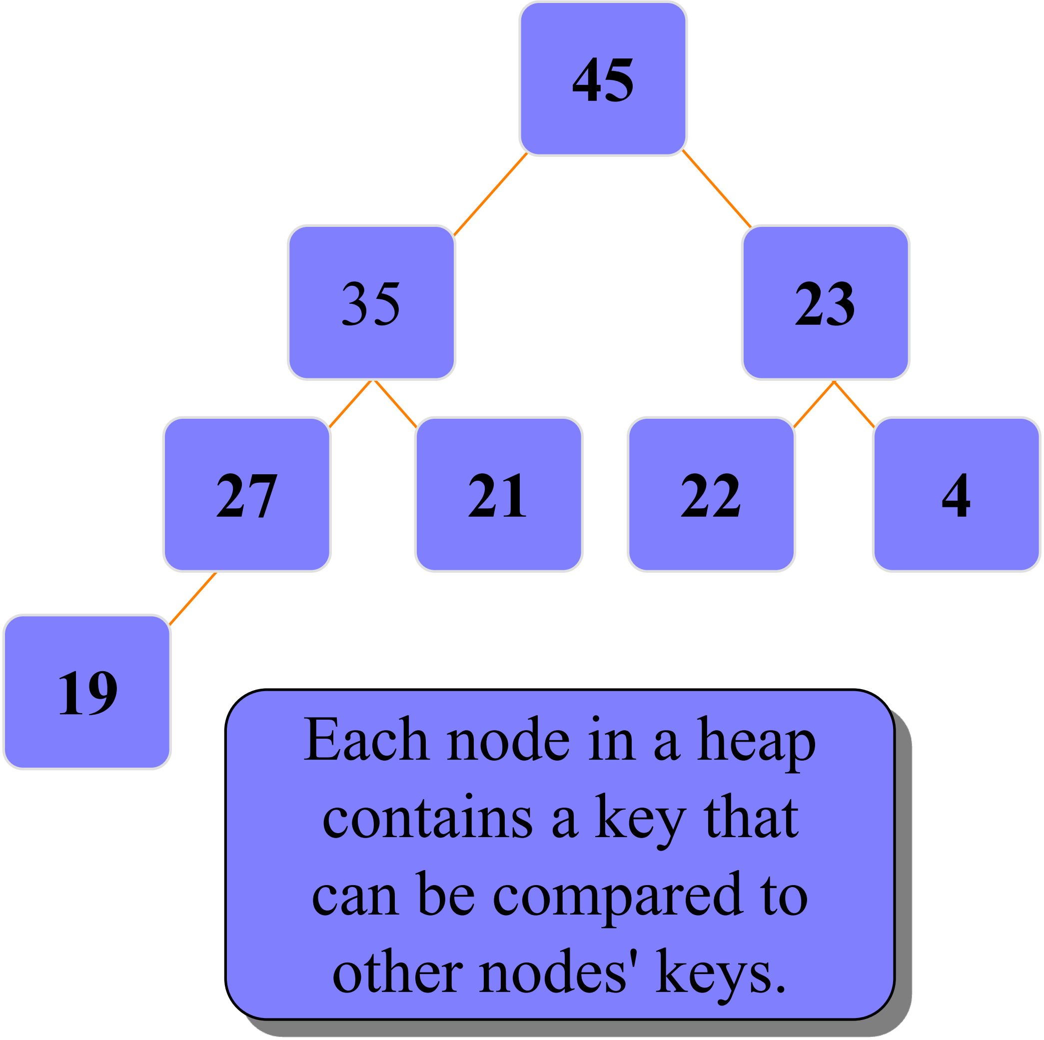
Heaps

Complete binary tree.



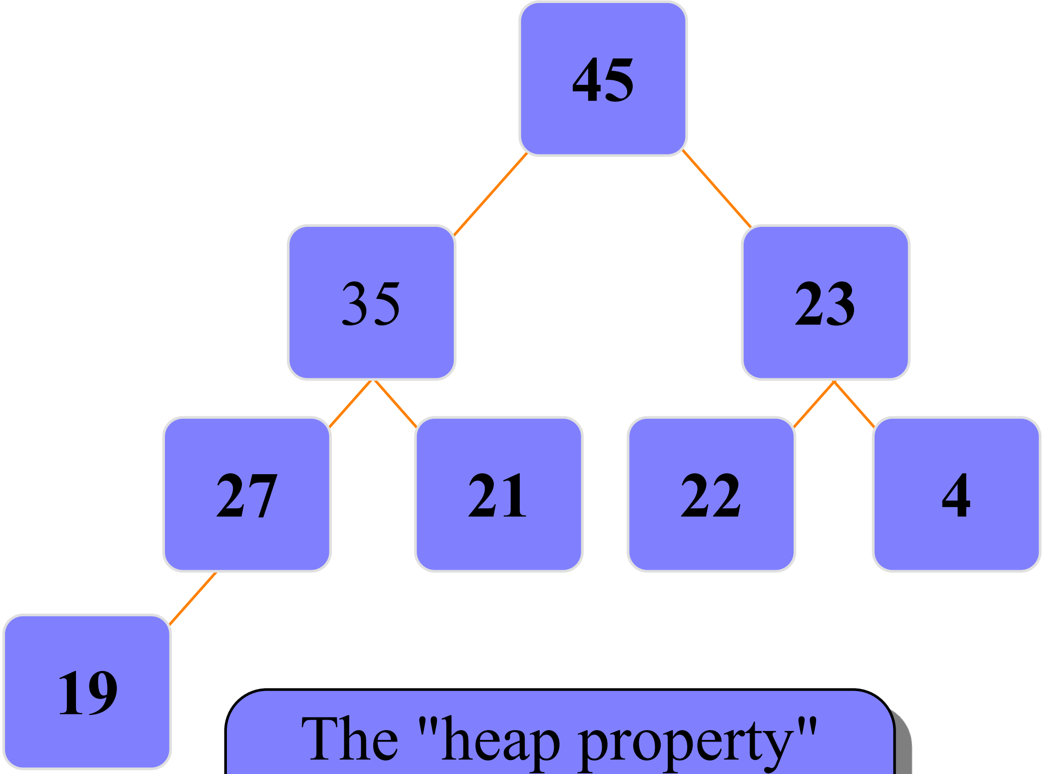
Heaps

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



Heaps

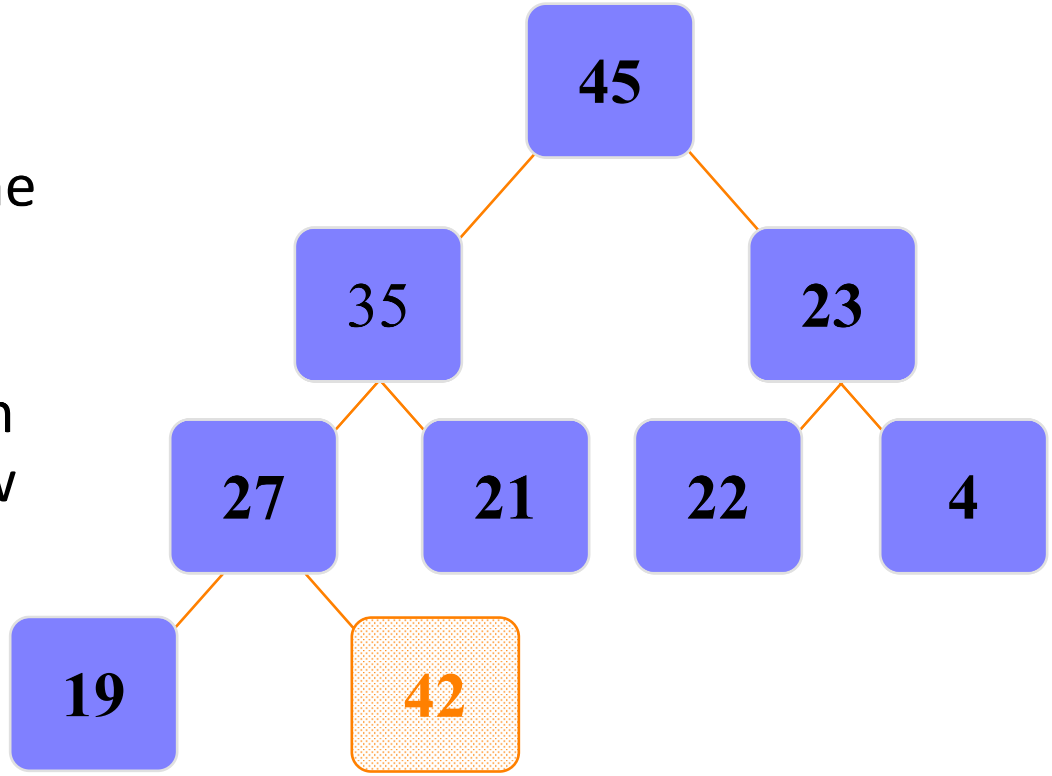
A heap is a **certain** kind of complete binary tree.



The "heap property" requires that each node's key is >= the keys of its children

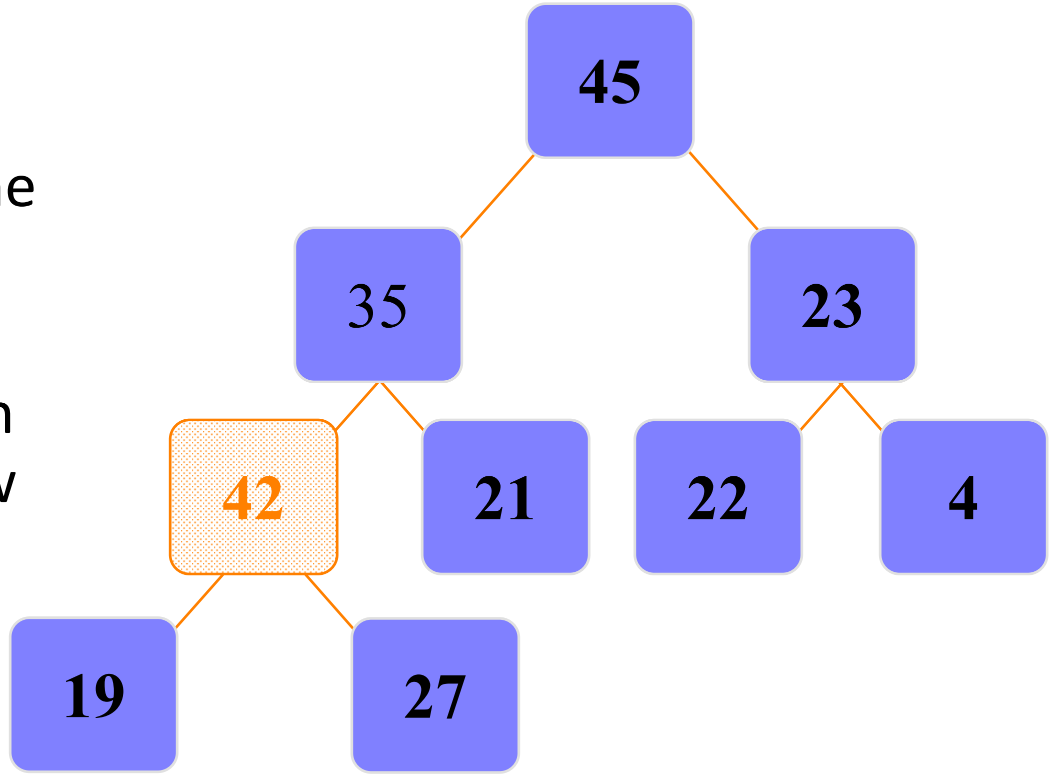
Adding a Node to a Heap

- 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.



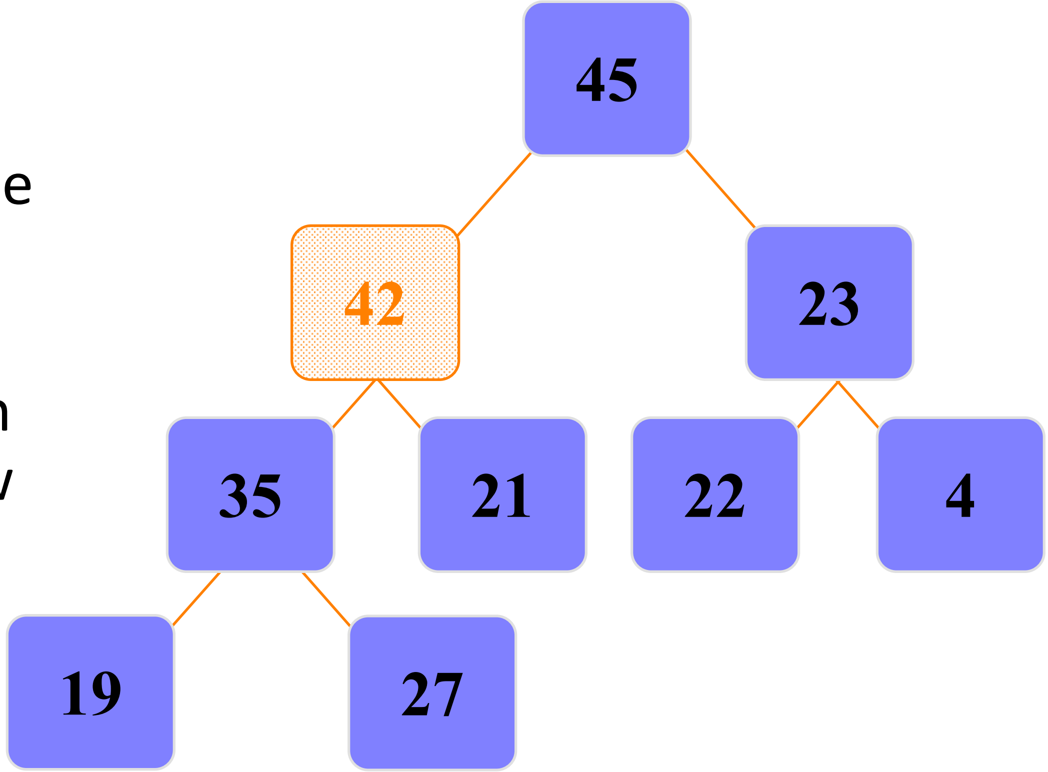
Adding a Node to a Heap

- 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.



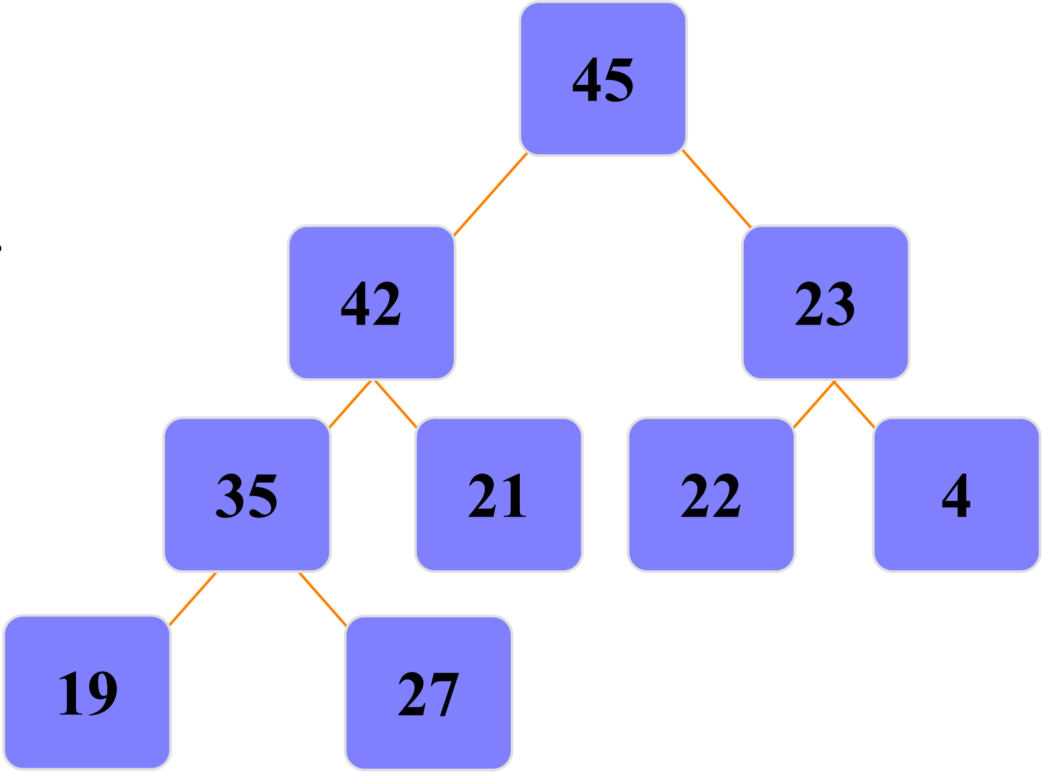
Adding a Node to a Heap

- 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.



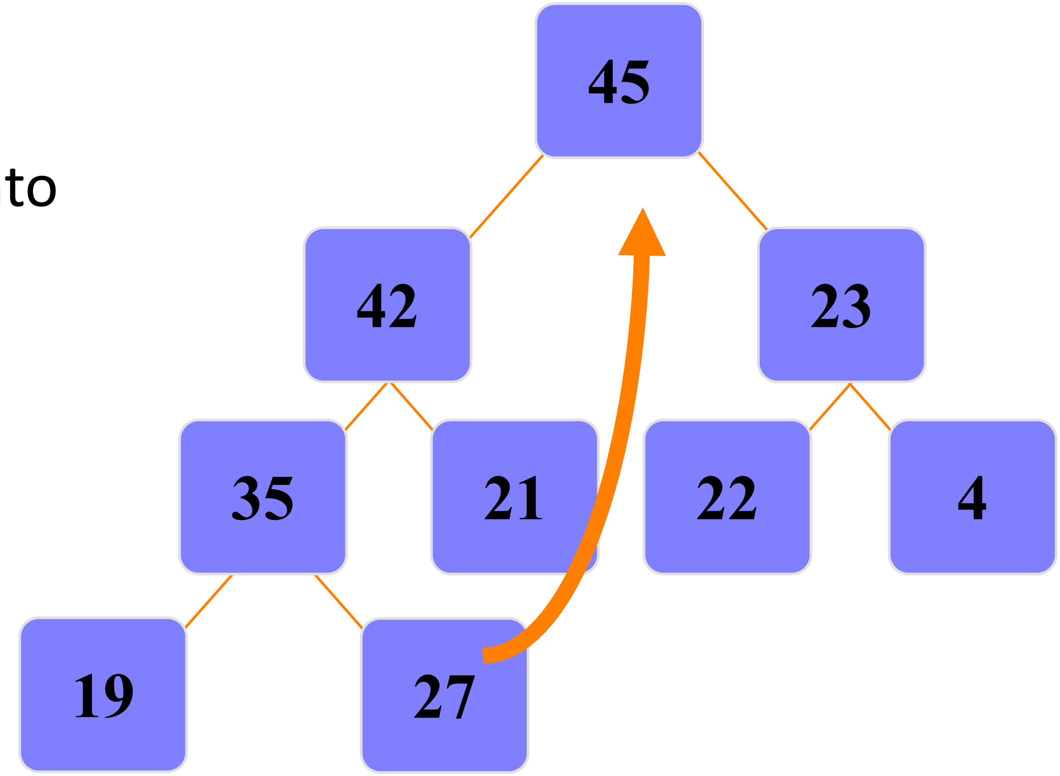
Adding a Node to a Heap

- 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**.



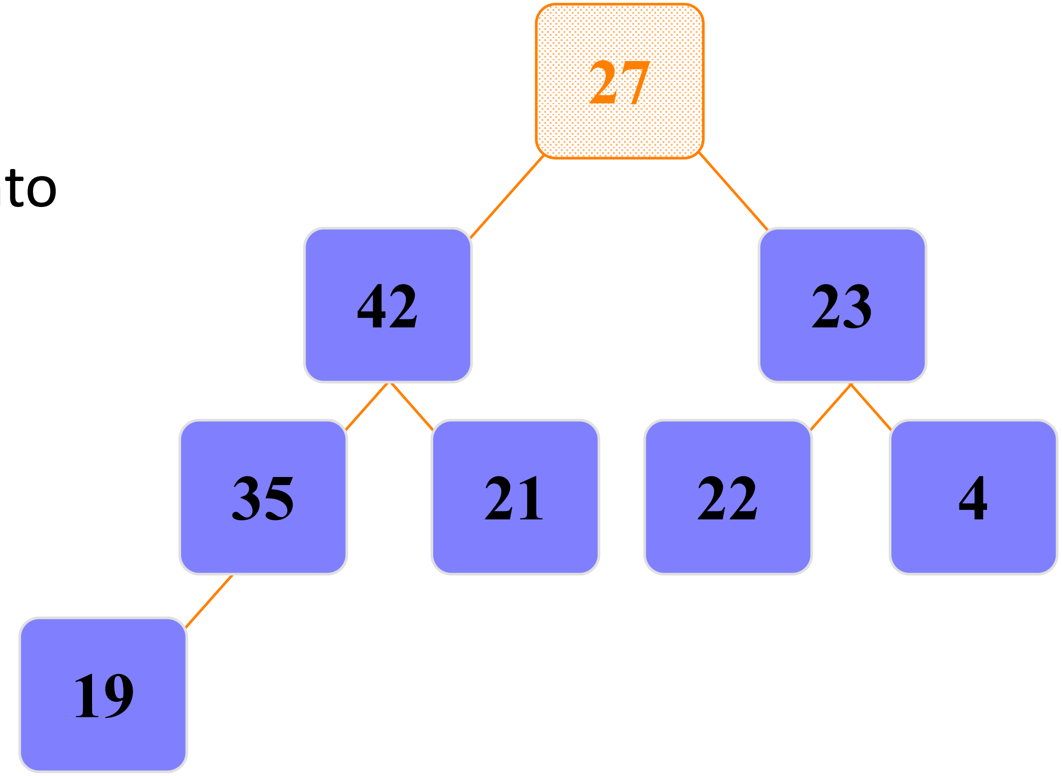
Removing the Top of a Heap

- Move the last node onto the root.



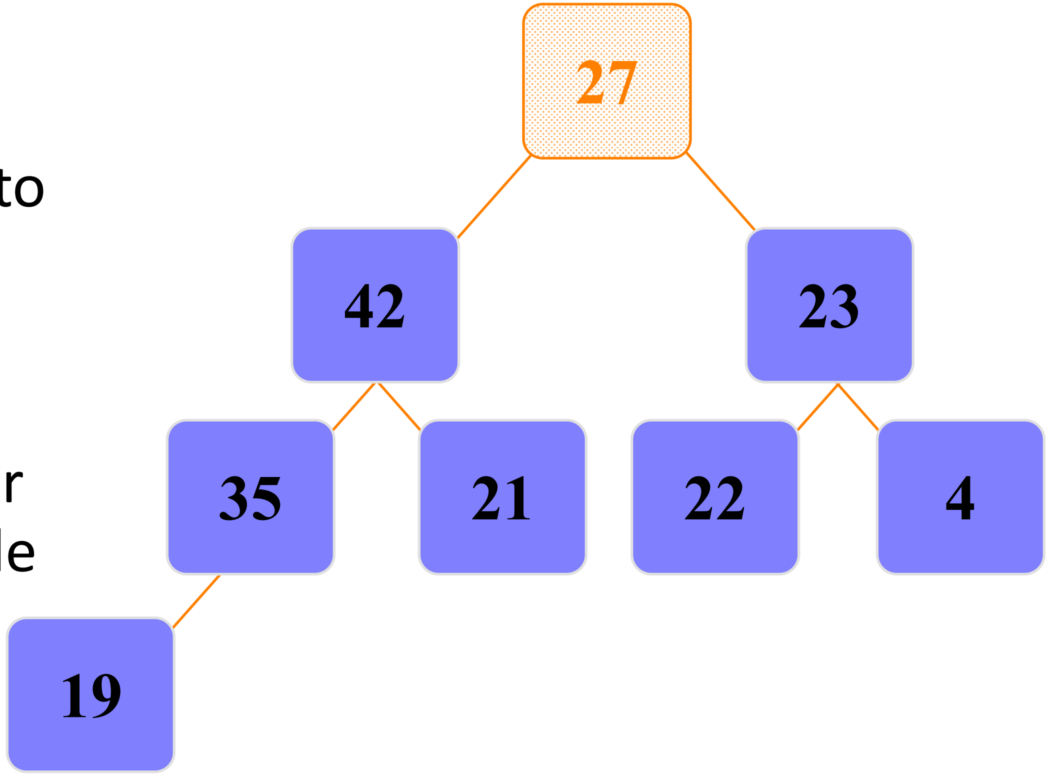
Removing the Top of a Heap

- Move the last node onto the root.



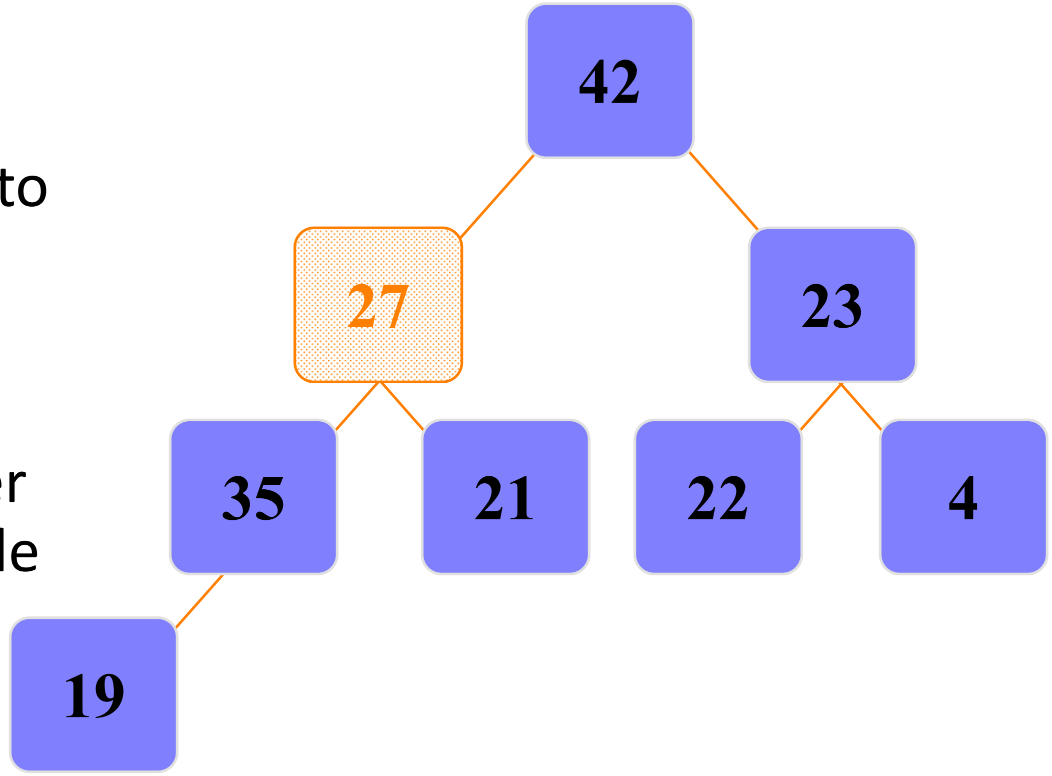
Removing the Top of a Heap

- ❑ 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.



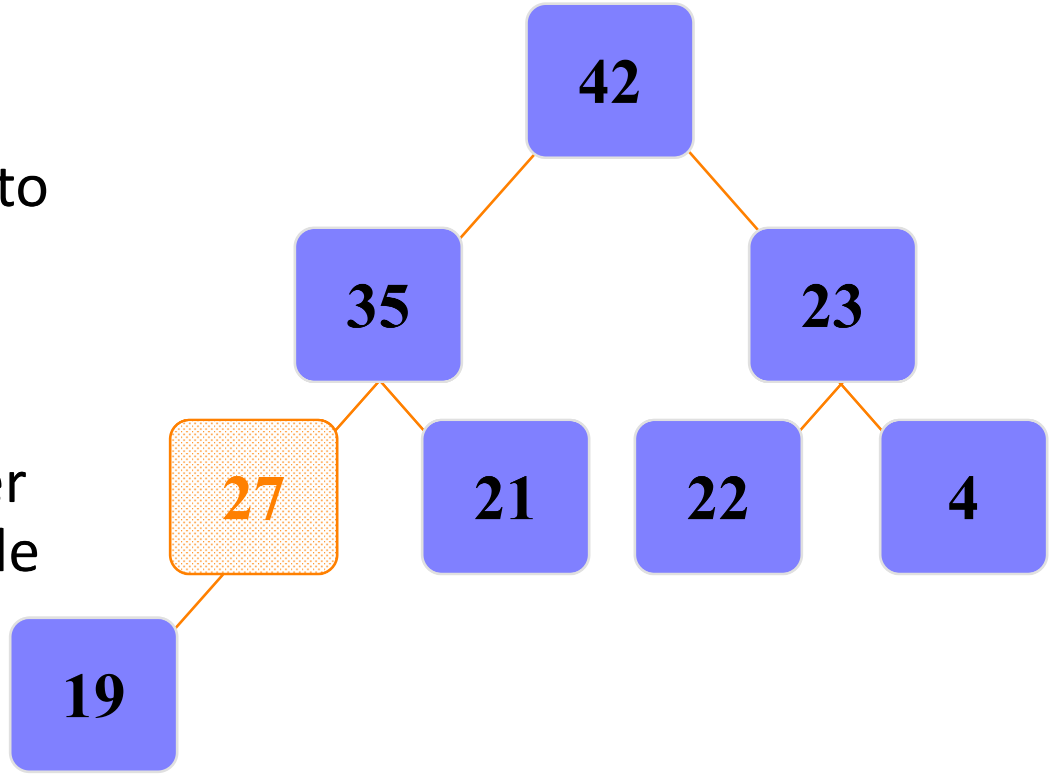
Removing the Top of a Heap

- 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.



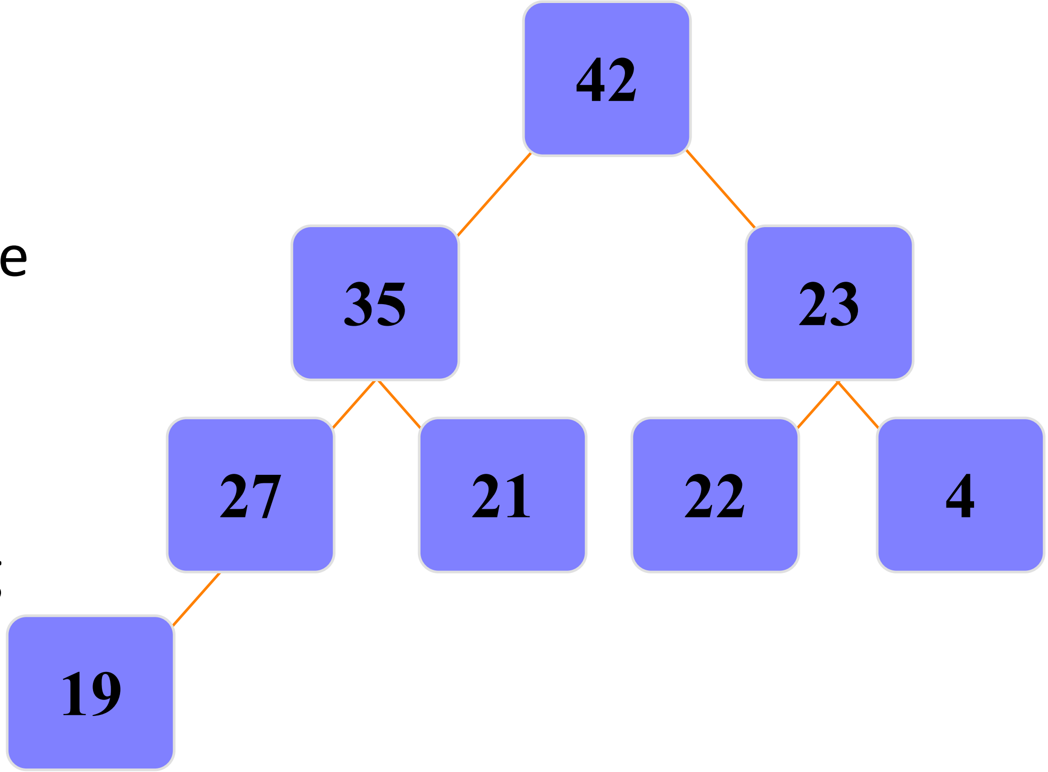
Removing the Top of a Heap

- ❑ 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.



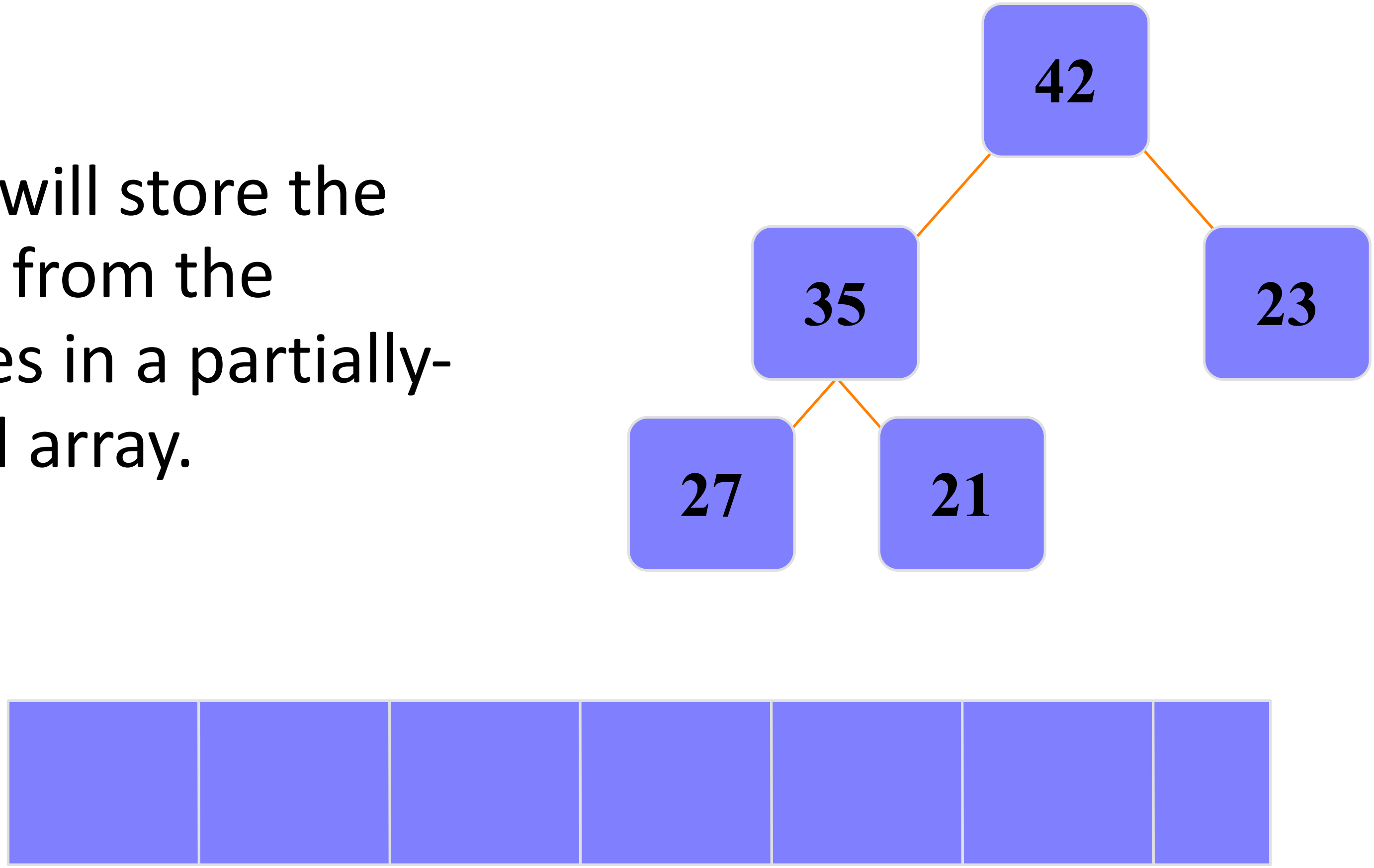
Removing the Top of a Heap

- 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**.



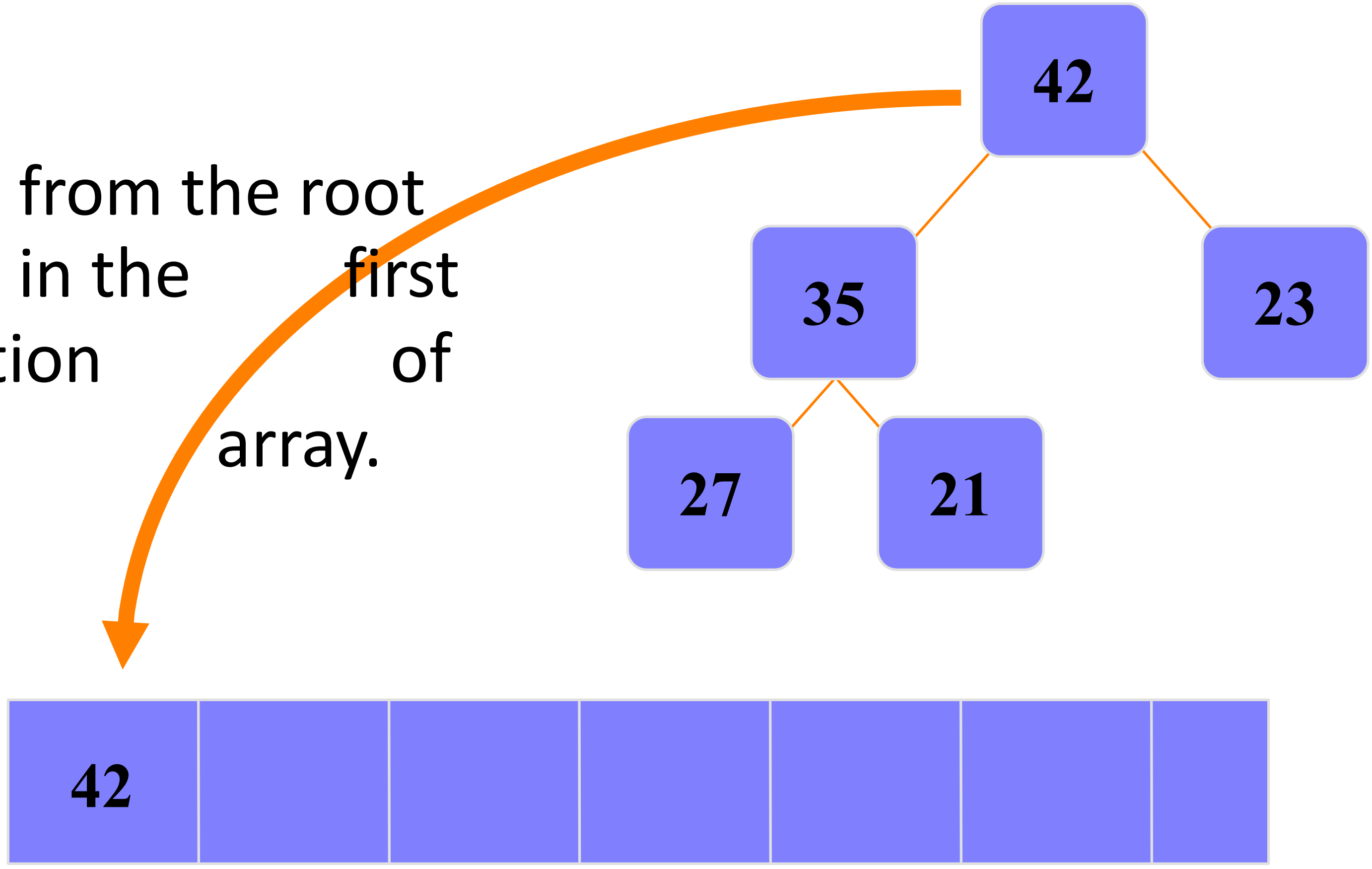
Implementing a Heap

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



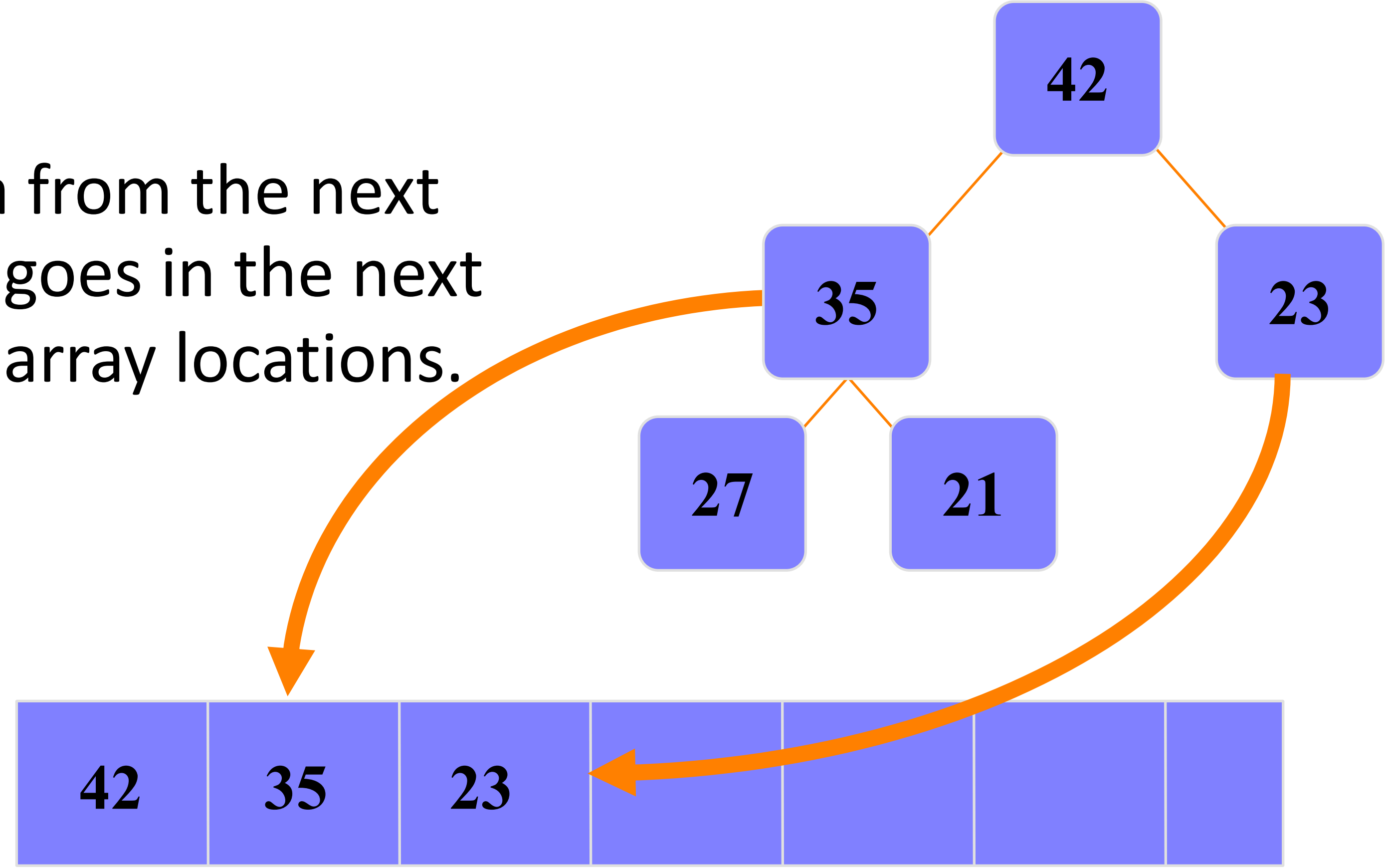
Implementing a Heap

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



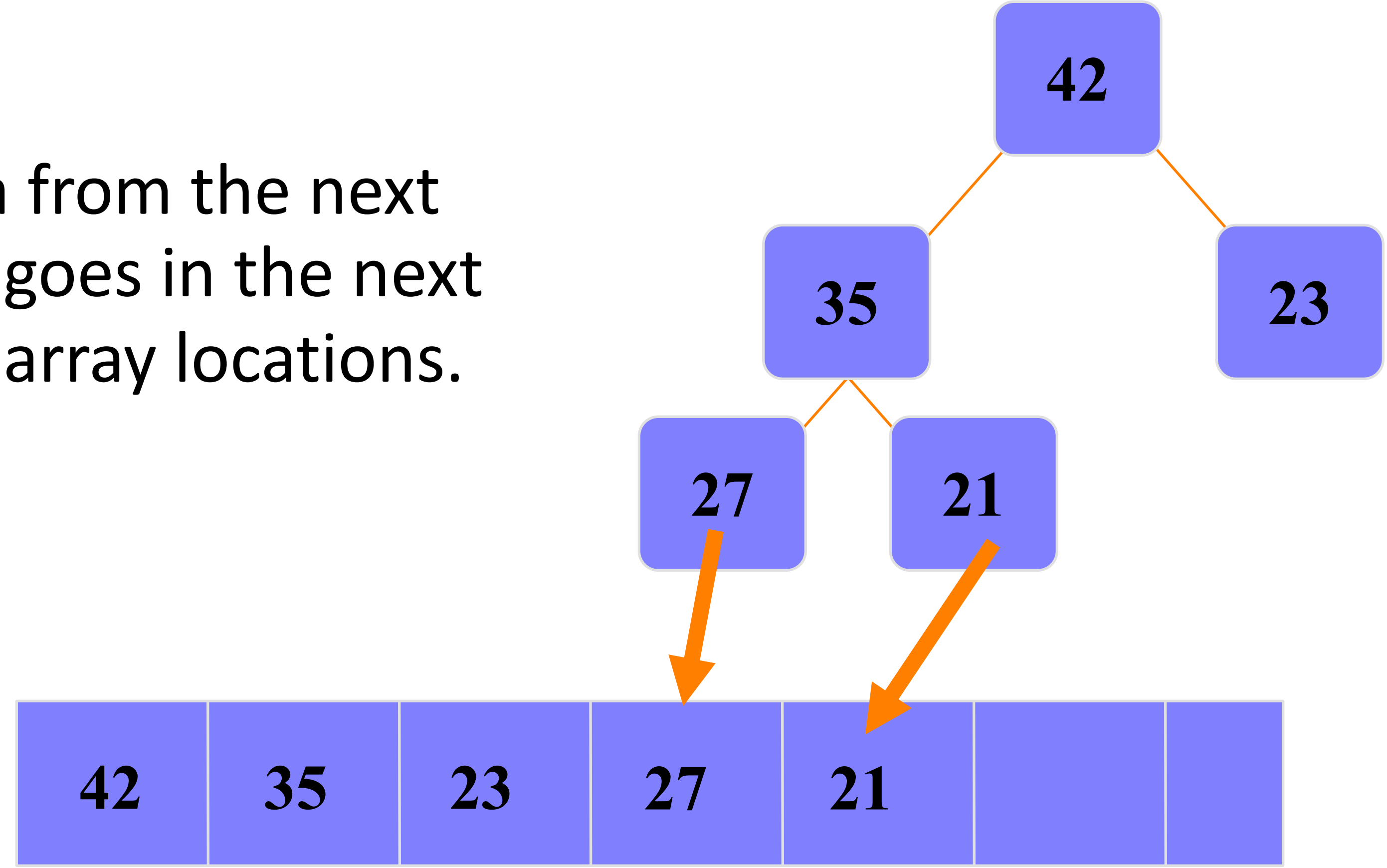
Implementing a Heap

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



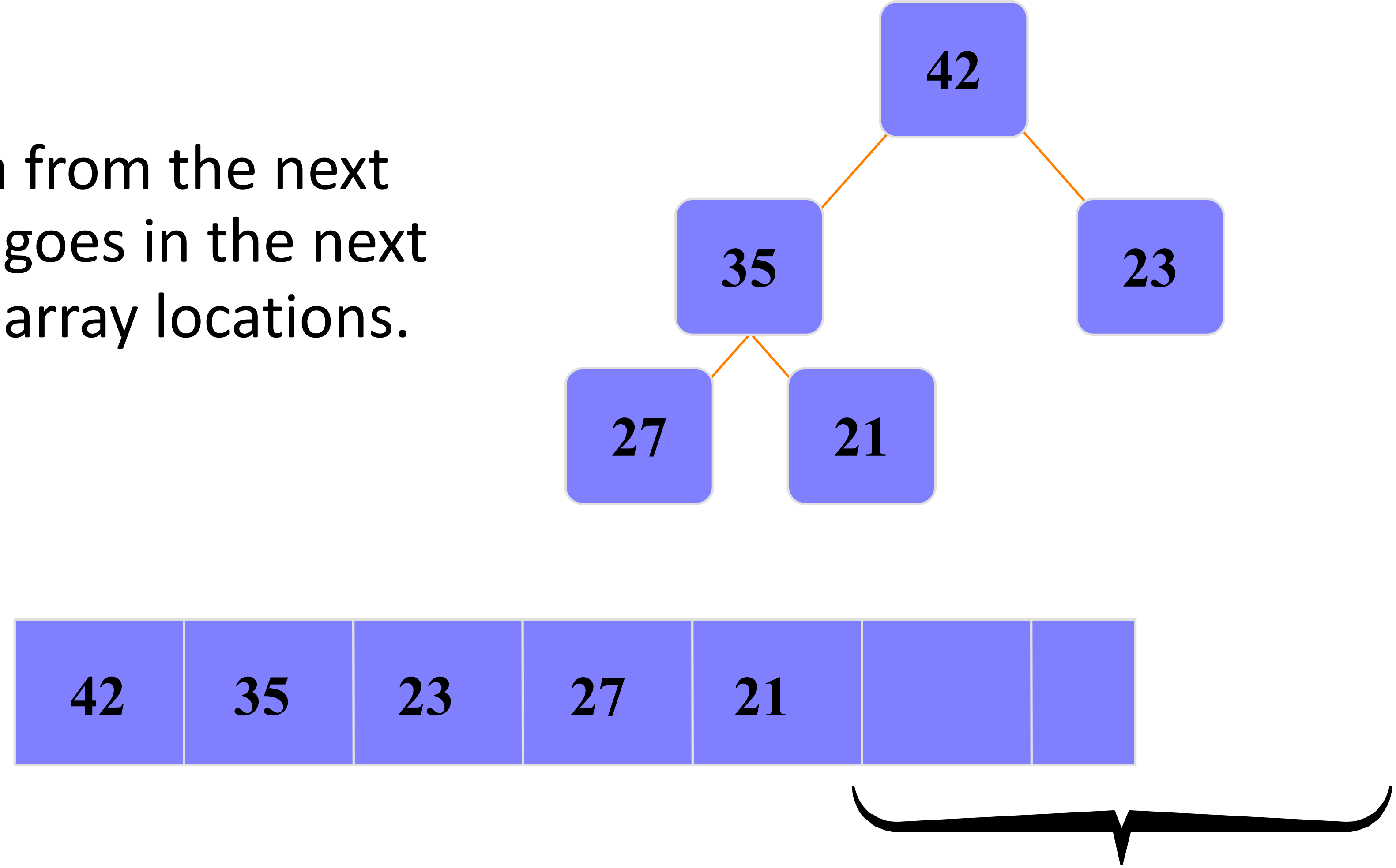
Implementing a Heap

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



Implementing a Heap

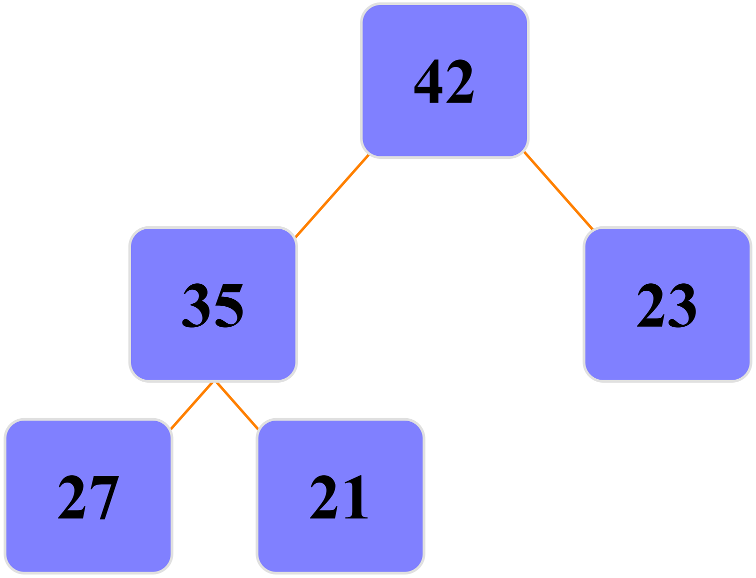
- Data from the next row goes in the next two array locations.



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

Important Points about the Implementation

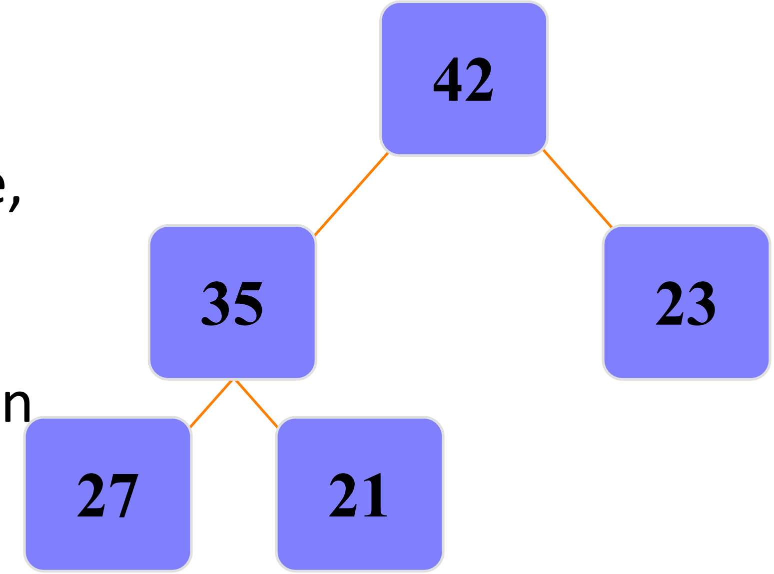
- The links between the tree's nodes are not actually stored as pointers, or in any other way.
- The only way we "know" that "the array is a tree" is from the way we manipulate the data.



42	35	23	27	21		

Important Points about the Implementation

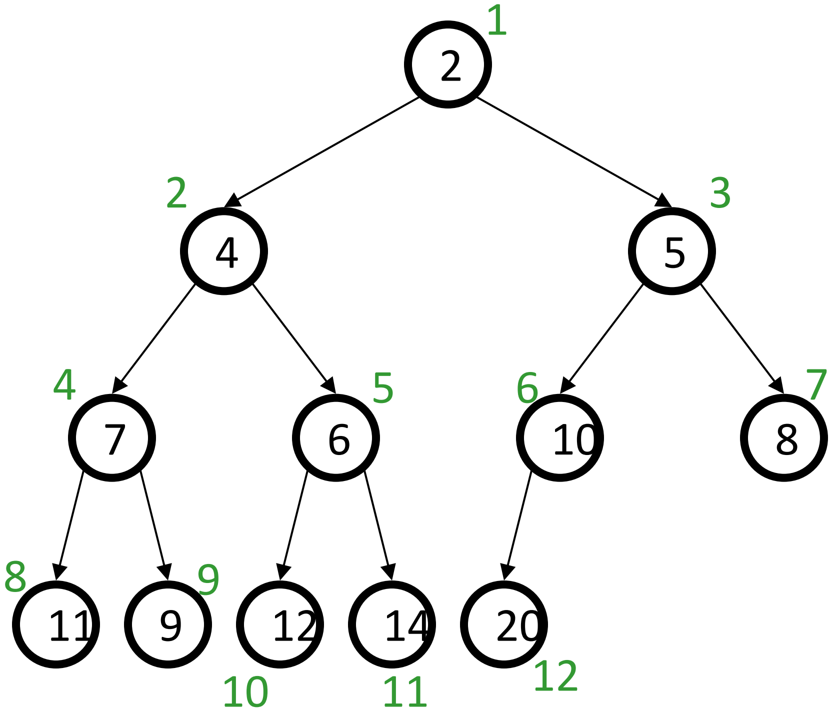
- If you know the index of a node, then it is easy to figure out the indexes of that node's parent and children. Formulas are given in the book.



42	35	23	27	21		

Nifty Storage Trick

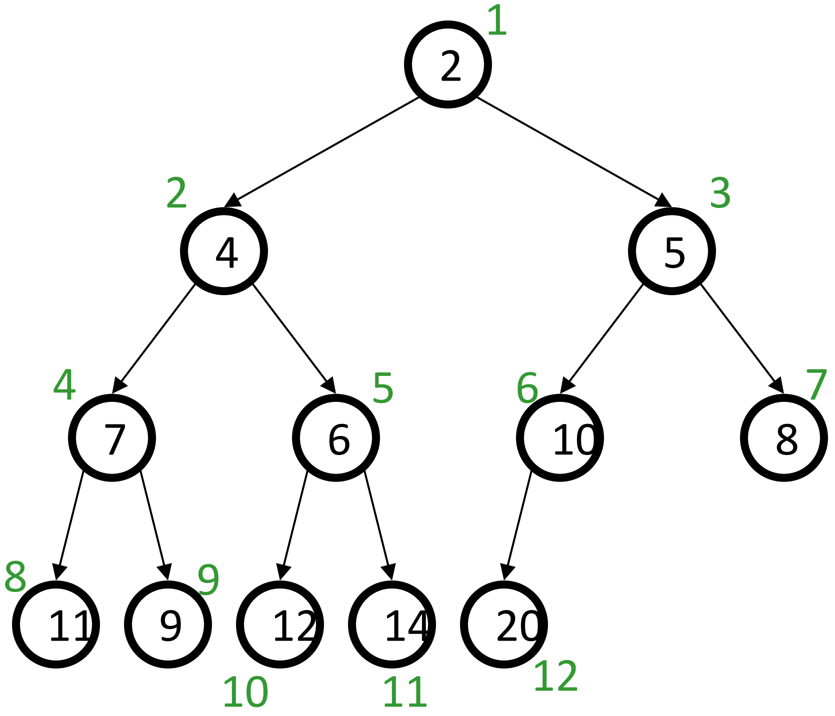
- Calculations:
  - child:
  - parent:
  - root:
  - next free:



0	1	2	3	4	5	6	7	8	9	10	11	12	
12	2	4	5	7	6	10	8	11	9	12	14	20	

Nifty Storage Trick

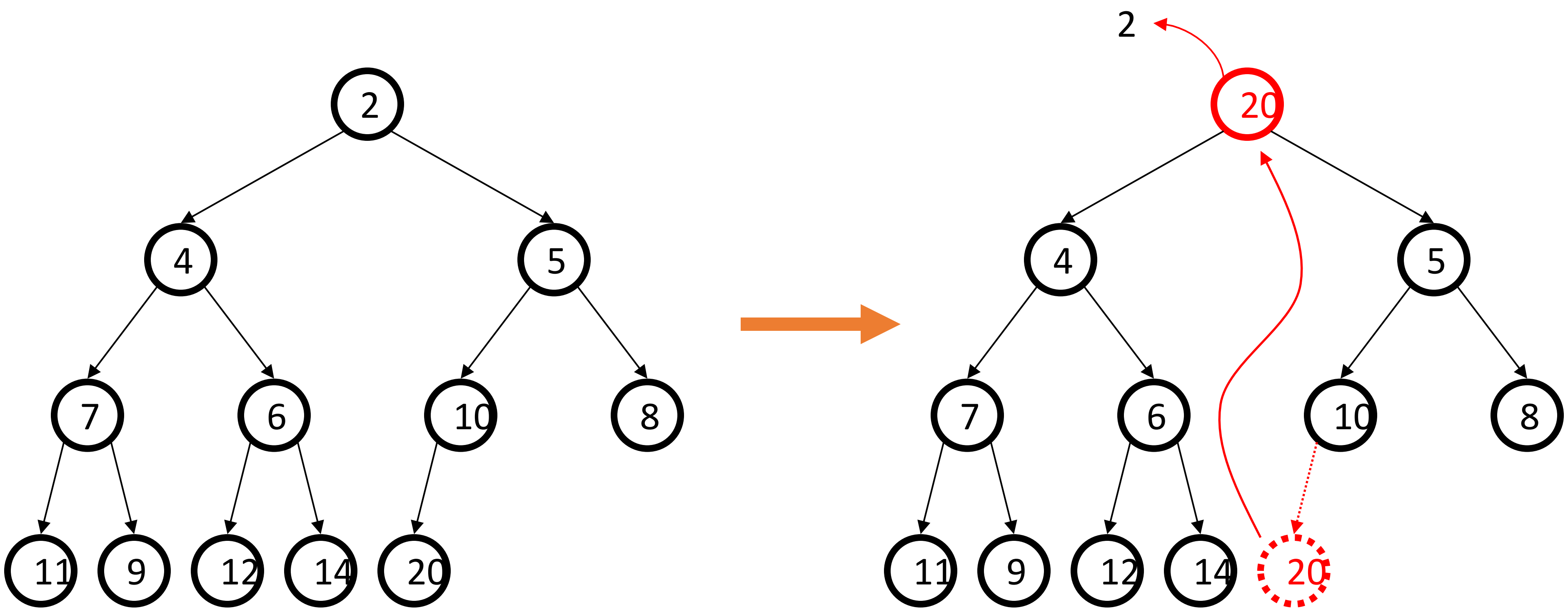
- Calculations:
  - child: left = 2*node
    right=2*node+1
  - parent: floor(node/2)
  - root: 1
  - next free: length+1



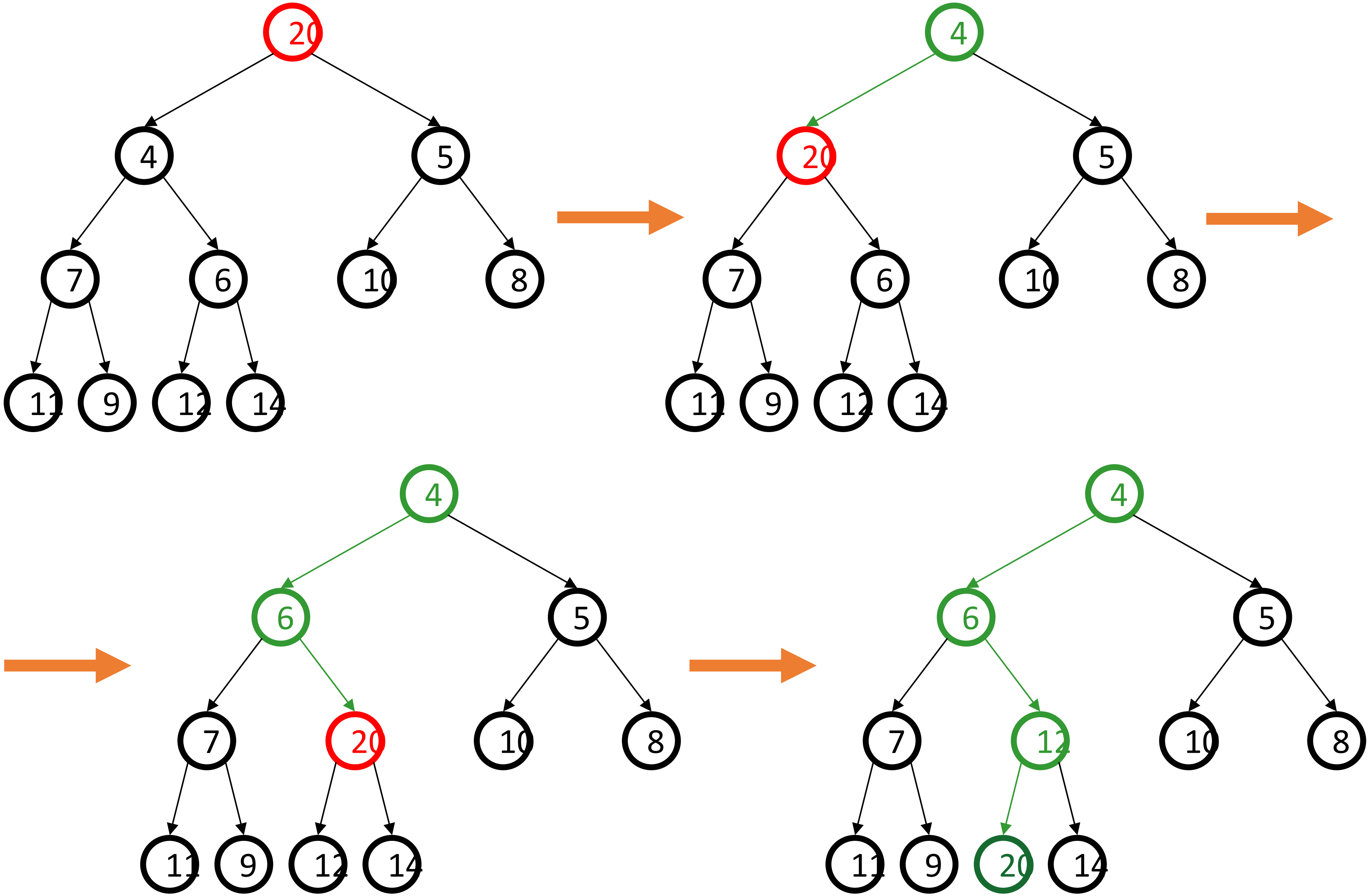
0	1	2	3	4	5	6	7	8	9	10	11	12	
12	2	4	5	7	6	10	8	11	9	12	14	20	

DeleteMin

`pqueue.deleteMin()`



Percolate Down



DeleteMin Code

```
Comparable deleteMin(){
  x = A[1];
  A[1]=A[size--];
  percolateDown(1);
  return x;
}
```

Trick to avoid repeatedly copying the value at A[1]

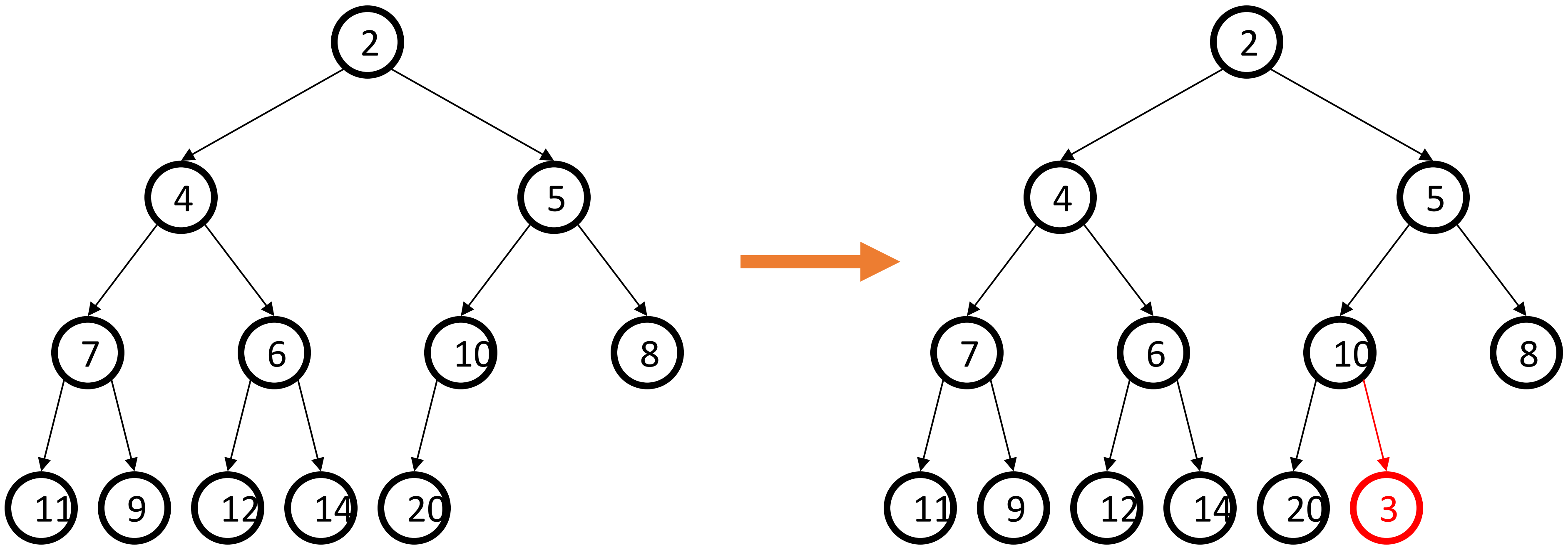
```
percolateDown(int hole) {
  tmp=A[hole];
  while (2*hole <= size) {
    left = 2*hole;
    right = left + 1;
    if (right <= size &&
         A[right] < A[left])
      target = right;
    else
      target = left;
    if (A[target] < tmp) {
      A[hole] = A[target];
      hole = target;
    }
    else
      break;
  }
  A[hole] = tmp;
}
```

Move down

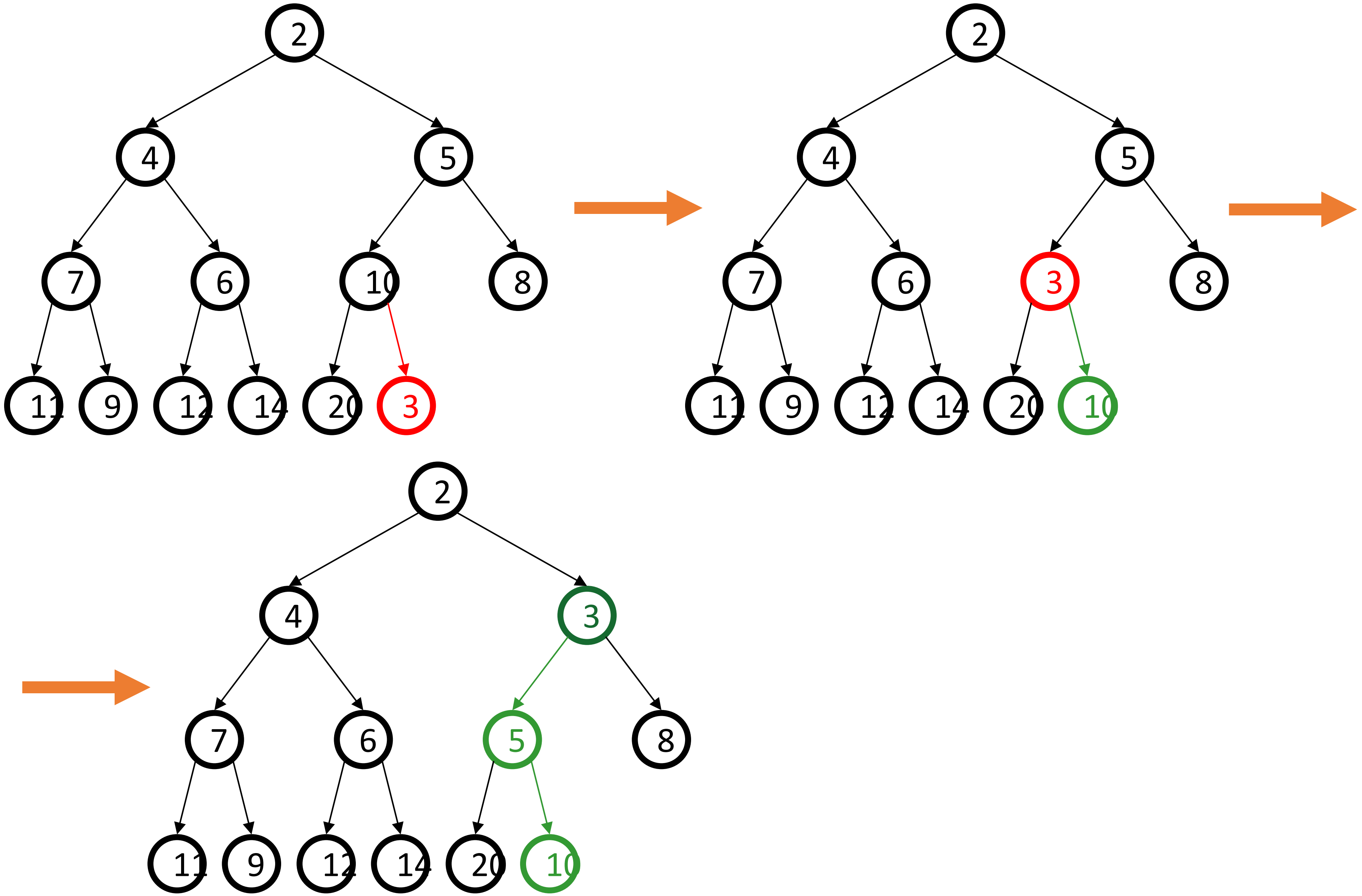
runtime:

Insert

pqueue.insert(3)



Percolate Up



Insert Code

```
void insert(Comparable x) {
  // Efficiency hack: we won't actually put x
  // into the heap until we've located the position
  // it goes in.  This avoids having to copy it
  // repeatedly during the percolate up.
  int hole = ++size;
  // Percolate up
  for( ; hole>1 && x < A[hole/2] ; hole = hole/2)
      A[hole] = A[hole/2];
  A[hole] = x;
}
```

runtime:

Performance of Binary Heap

	Binary heap worst case	Binary heap avg case	AVL tree worst case	BST tree avg case
Insert	O(log n)	O(1) percolates 1.6 levels	O(log n)	O(log n)
Delete Min	O(log n)	O(log n)	O(log n)	O(log n)

- In practice: binary heaps much simpler to code, lower constant factor overhead