CPSC 221
Basic Algorithms and Data Structures

June 8, 2015

Administrative stuff

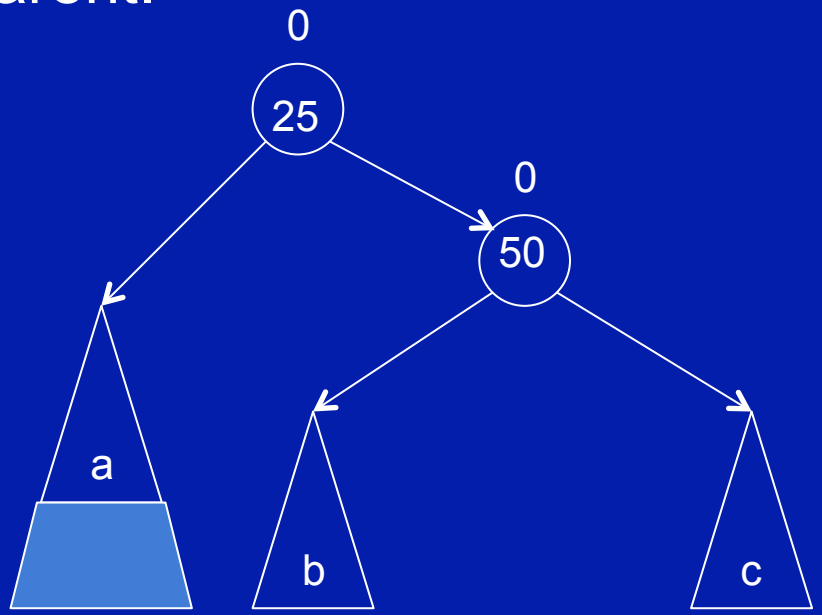
Exam marking still not quite done.

Labs resume tomorrow. Lab 6 is posted.

Theory assignment 2 is posted. Programming assignment 2 waits until Wednesday's lecture.

AVL trees

The AVL algorithm looks for four kinds of unbalanced trees: The Left-Left tree (parent and child nodes are both left-heavy, parent balance is -2, child balance is -1). Fix by rotating right around the parent.



Balanced, non-binary trees

Not all search trees have to be binary trees to be useful.

If we increase the branching factor in our search trees from 2 to m, the worst and average case search times can be improved to $O(\log_m n)$ comparisons (instead of $O(\log_2 m)$ comparisons). In really big file systems, that's a savings worth pursuing.

Increasing the branching factor of a search tree greatly reduces the number of levels which must be searched for a given key.

B-trees

One type of m-ary tree is the B-tree.

As your book says:

"B-trees were developed to store indexes to databases stored on disk. Disk storage is broken into blocks, and the time to access a block is significant compared to the time required to manipulate the data once it is in internal memory. The nodes of a B-tree are sized to fit in a block, so each disk access to the index retrieves exactly one B-tree node.... [By] making the tree nodes as large as possible, we reduce the number of disk accesses required to find an item in the index. Assuming that a block can store a node for a B-tree of order 200, each node would store at least 100 items. This would enable 100^4, or 100 million, items to be accessed in a B-tree of height 4."

How you arrive at those numbers becomes more obvious once you read the chapter, but these are good numbers.

B-trees

B-tree properties:

A B-tree of order m is a tree with the following properties:

- the root has at least two children unless it is a leaf
- no node has more than m children
- every node except for the root and leaves has at least ceiling(m/2) children.
- all leaves appear on the same level
- an internal node with k children contains exactly k – 1 keys

“B” may stand for Bayer (one of its creators), Balanced, Broad, Bushy, or even Boeing (Bayer’s employer). but it does not stand for Binary.

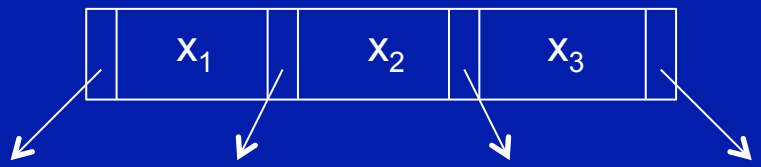
B-trees

What's in the nodes?

The literature is fuzzy and inconsistent, especially with respect to leaf nodes, but often the story goes like this:

An internal node in an order m B-tree contains an ordered set of as many as m – 1 keys and m child pointers.

The keys are "road signs" that direct traversal through the tree. A node containing keys x_1, x_2, x_3 will point to values < x_1 in its left-most subtree, >= x_1 and <x_2 in the subtree pointed to between x_1 and x_2, and so on.



B-trees

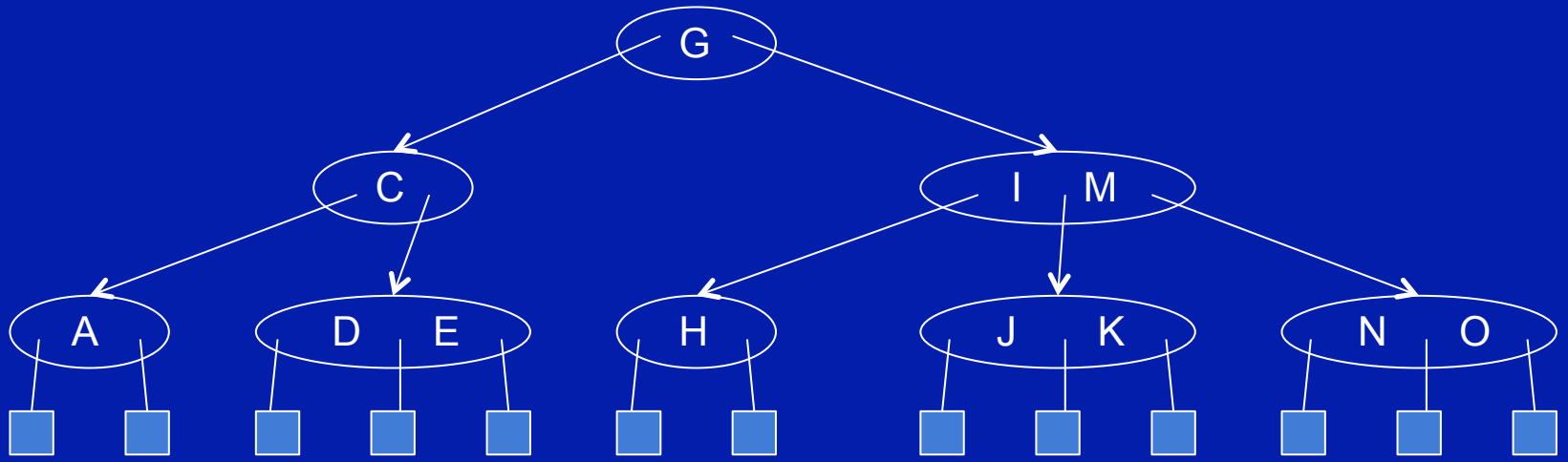
What's in the nodes?

A leaf node in an order m B-tree contains an ordered set of elements that represent data and no child pointers.

The elements may be the actual data itself or pointers to data.

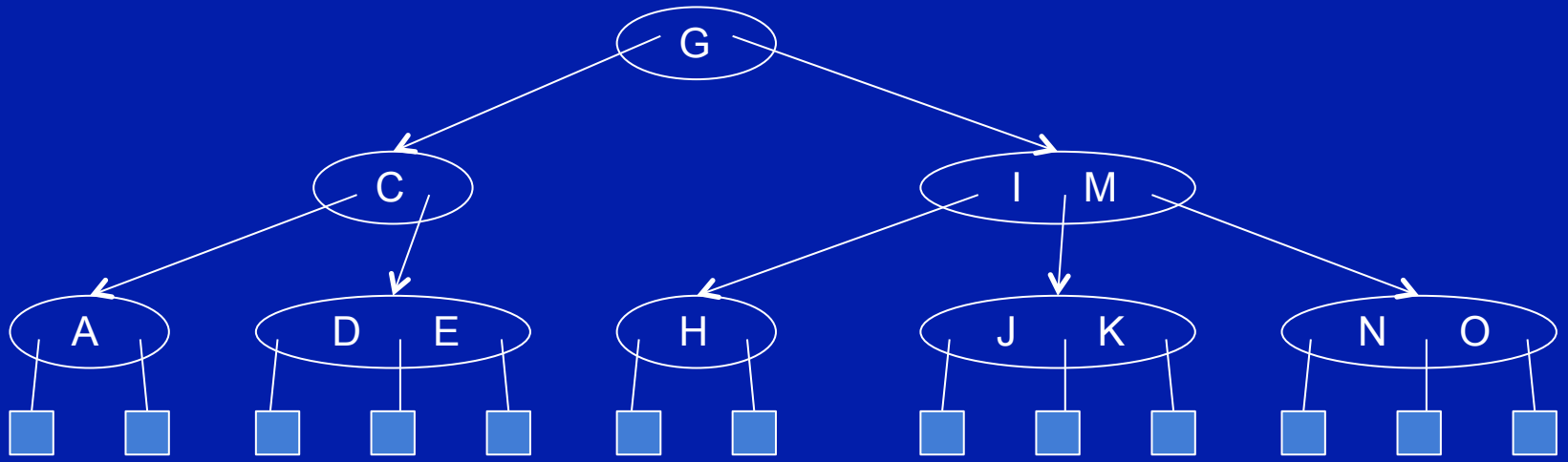
To keep things simple, we're not going to deal with the data, just like your book. We'll just talk about B-trees (and their variants) as if there were only key values and internal nodes, and we'll let the data be a detail that you deal with if/when you implement your own B-trees.

2-3 trees



A simple variant of the B-tree is the 2-3 tree. A 2-3 tree is a B-tree of order 3 (m = 3). Thus, every internal node in a 2-3 tree must have either two or three children, each internal node must contain one or two keys, and all the leaves must appear on one level.

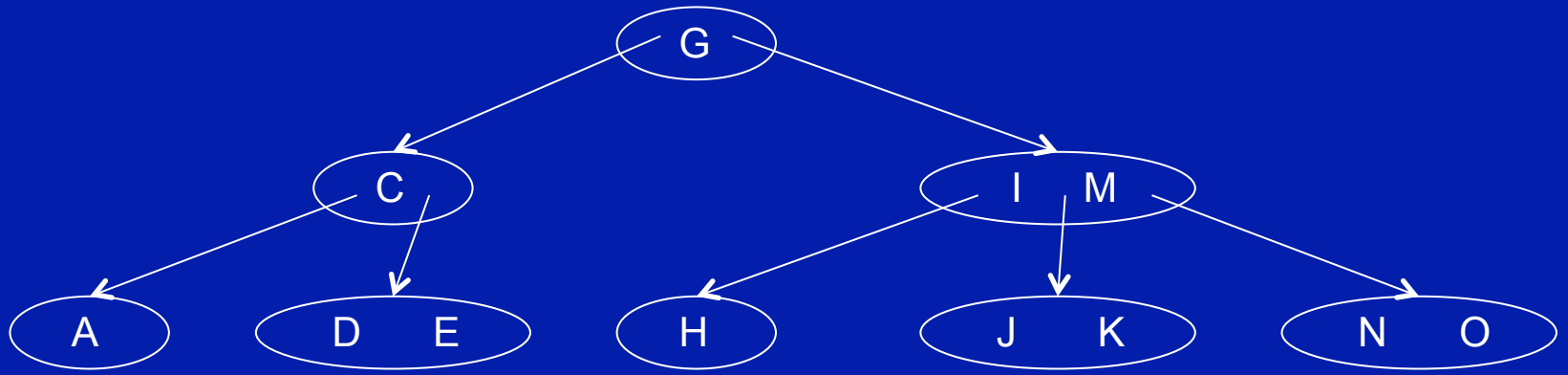
2-3 trees



A simple variant of the B-tree is the 2-3 tree. A 2-3 tree is a B-tree of order 3 (m = 3). Thus, every internal node in a 2-3 tree must have either two or three children, each internal node must contain one or two keys, and all the leaves must appear on one level.

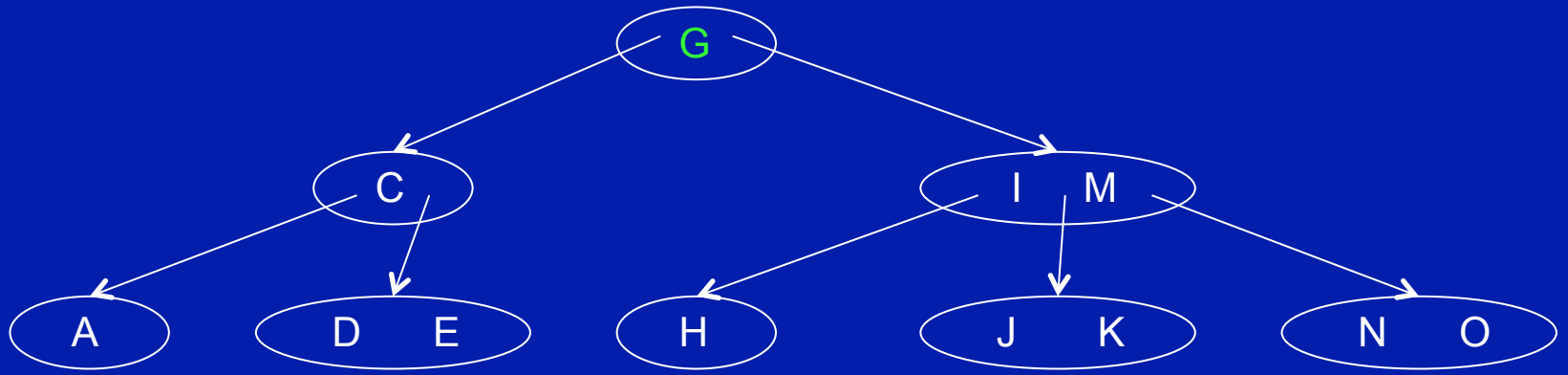
The leaves here are the little boxes. They hold the data. We'll now abstract the data away. Say goodbye.

2-3 trees



The keys are stored in order such that an inorder traversal visits the keys in sorted order, just like with a binary search tree.

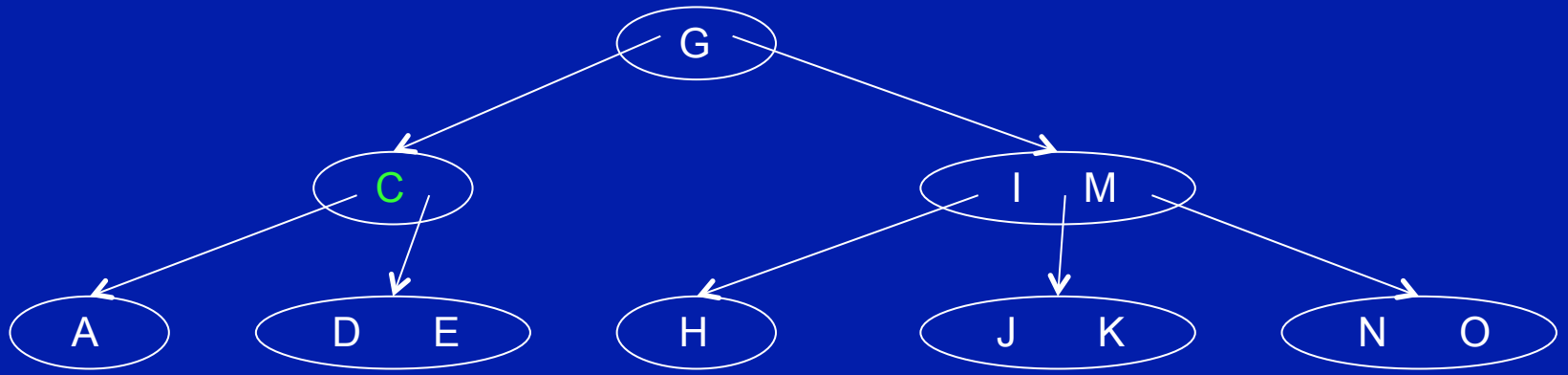
2-3 trees



Search is much like search in a binary search tree.

To search for key D
start at root

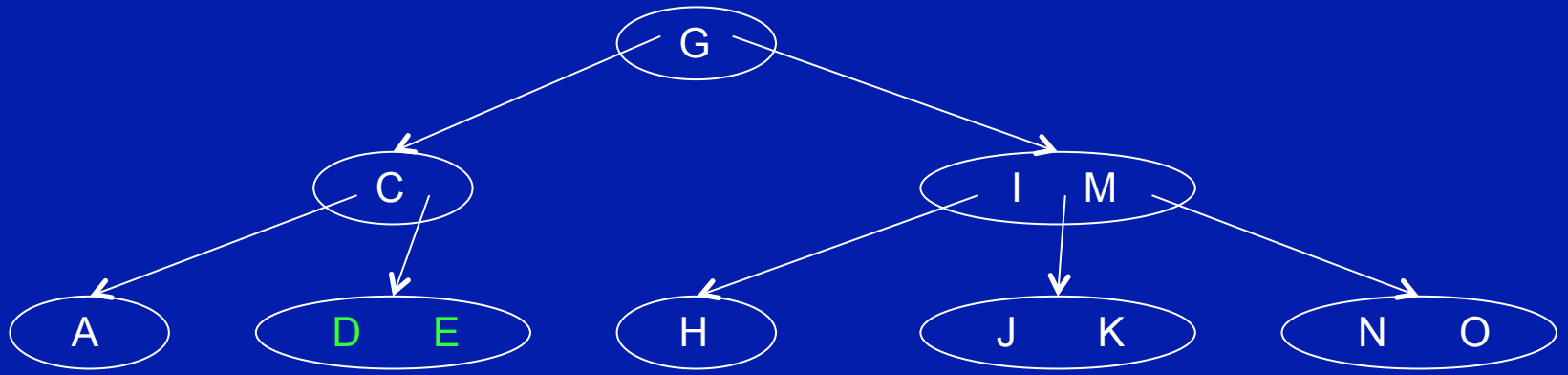
2-3 trees



Search is much like search in a binary search tree.

To search for key D
start at root
D < G so follow left pointer

2-3 trees



Search is much like search in a binary search tree.

To search for key D

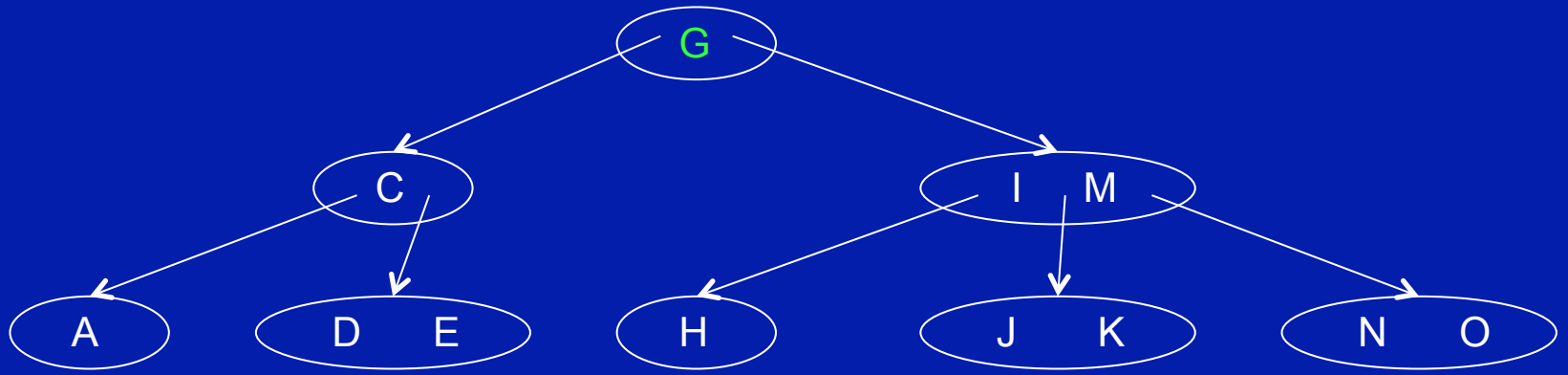
start at root

D < G so follow left pointer

D > C so follow right pointer

D is one of the keys at this node so success!

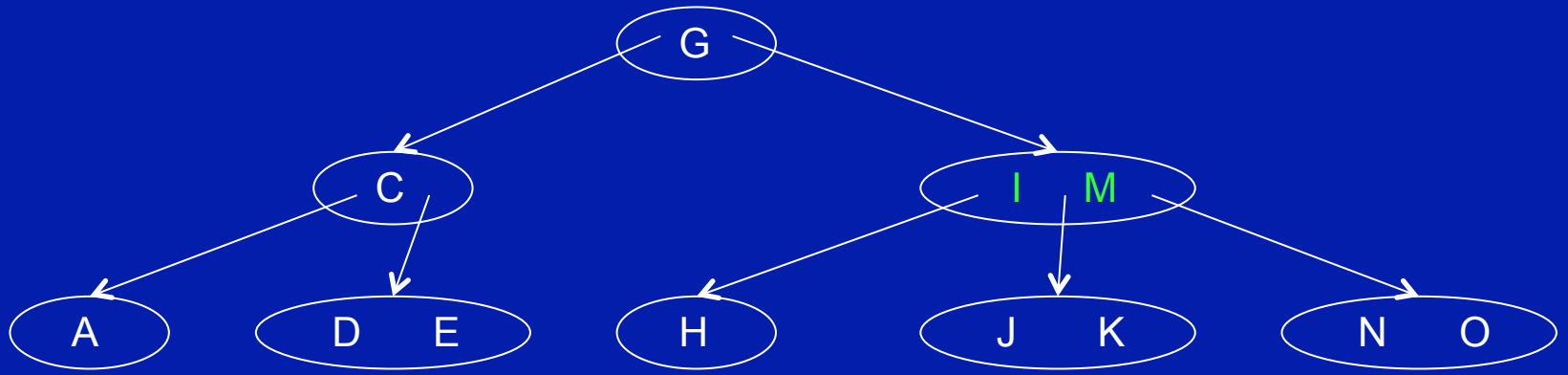
2-3 trees



Search is much like search in a binary search tree.

To search for key L
start at root

2-3 trees



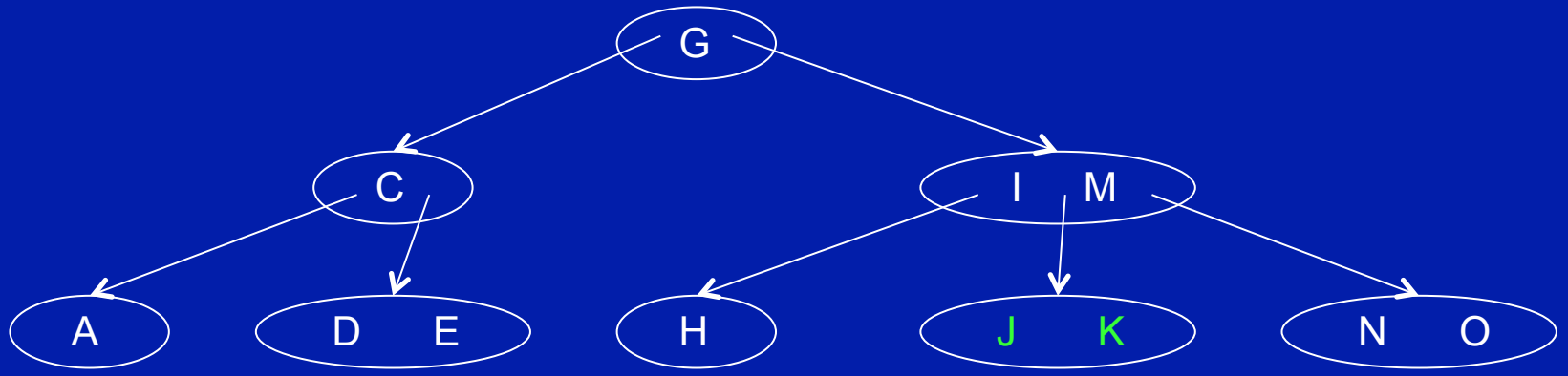
Search is much like search in a binary search tree.

To search for key L

start at root

L > G so follow right pointer

2-3 trees



Search is much like search in a binary search tree.

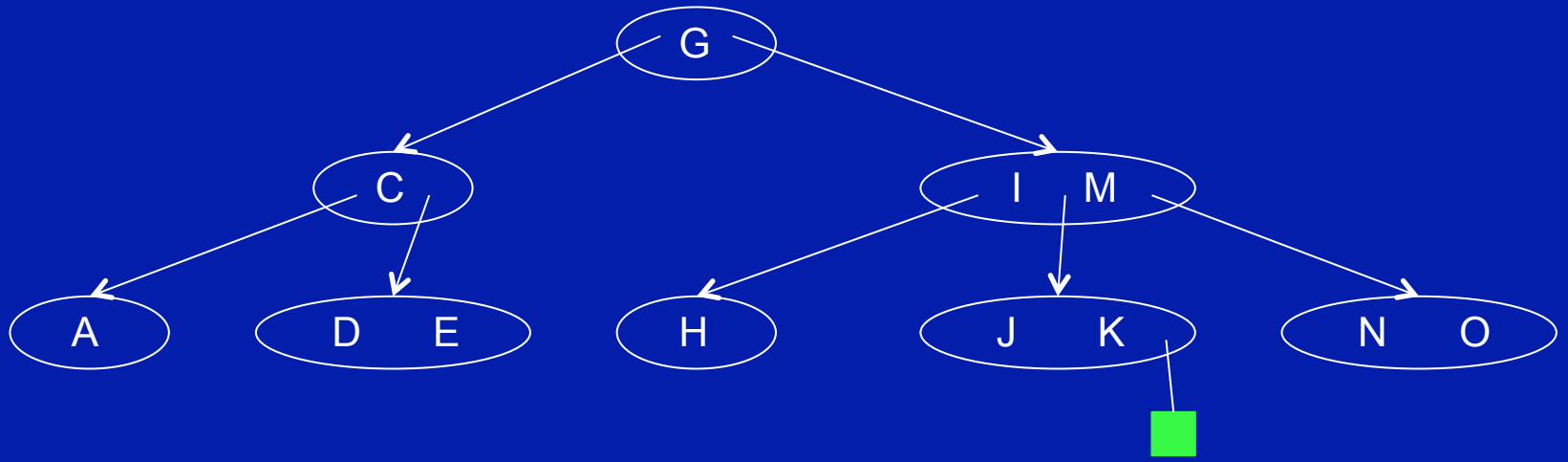
To search for key L

start at root

L > G so follow right pointer

I < L < M so follow middle pointer

2-3 trees

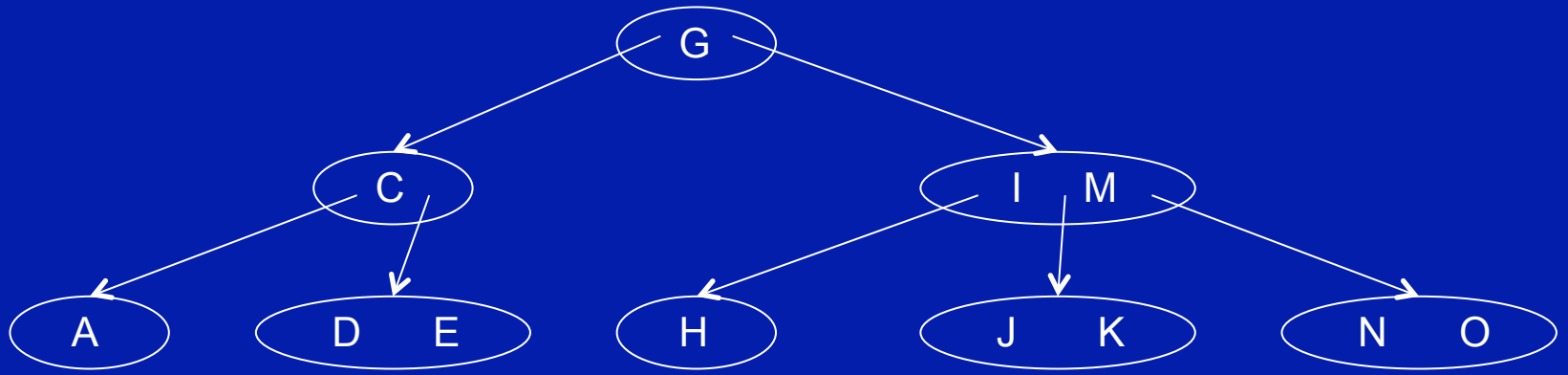


Search is much like search in a binary search tree.

To search for key L

start at root
L > G so follow right pointer
I < L < M so follow middle pointer
L > K so follow right pointer, but that gets us to data – key not found!

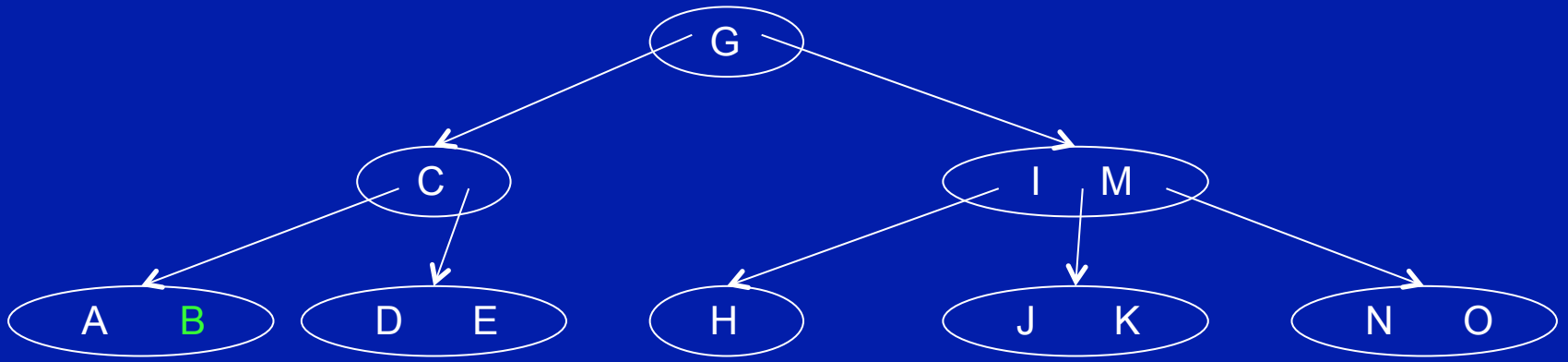
2-3 trees



Insertion is fairly simple

To insert key B, we just search for B and add B to the node where it should be.

2-3 trees

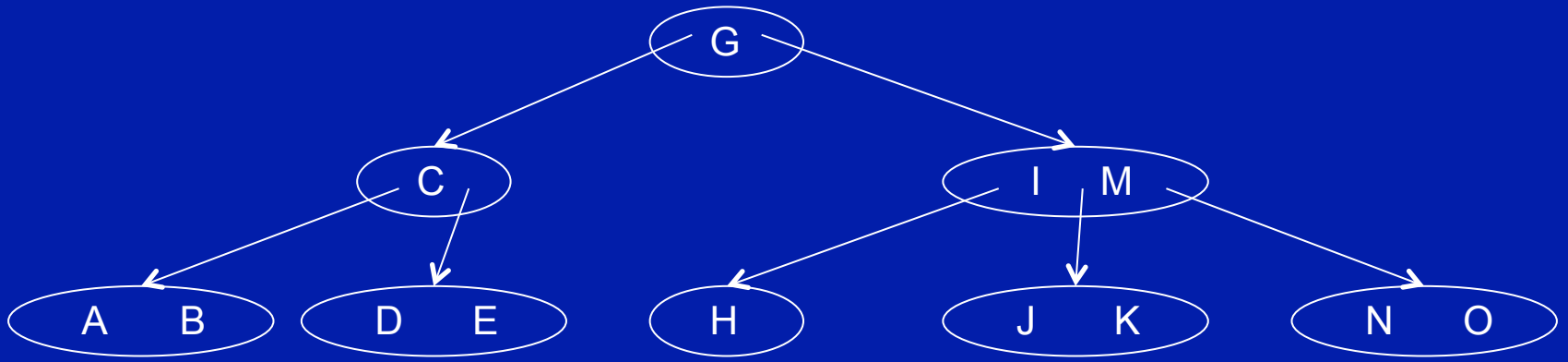


Insertion is fairly simple

To insert key B, we just search for B and add B to the node where it should be.

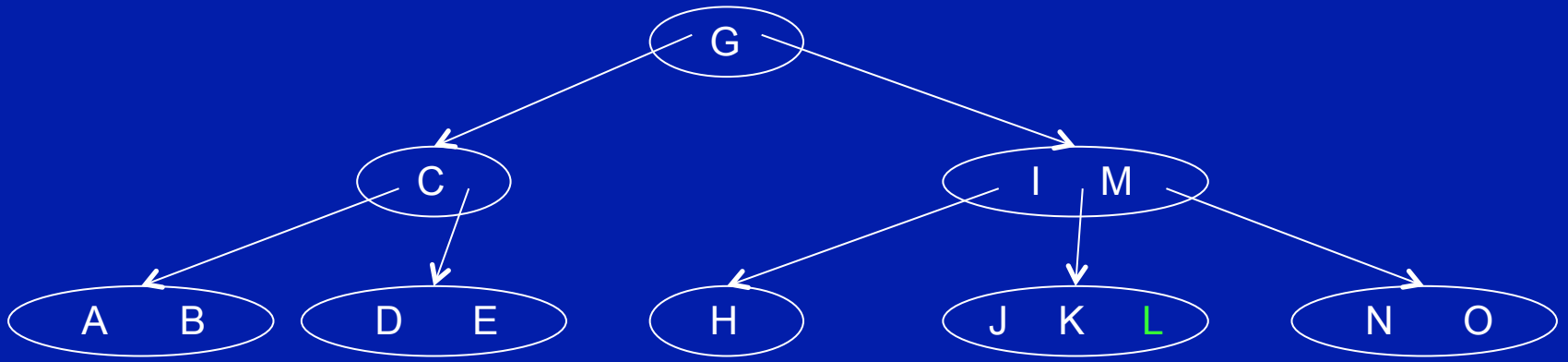
Depth of tree remains the same. The number of keys in a node is increased. All the B-tree properties are preserved.

2-3 trees



To insert key L, again we search to find the node where it should be and add it.

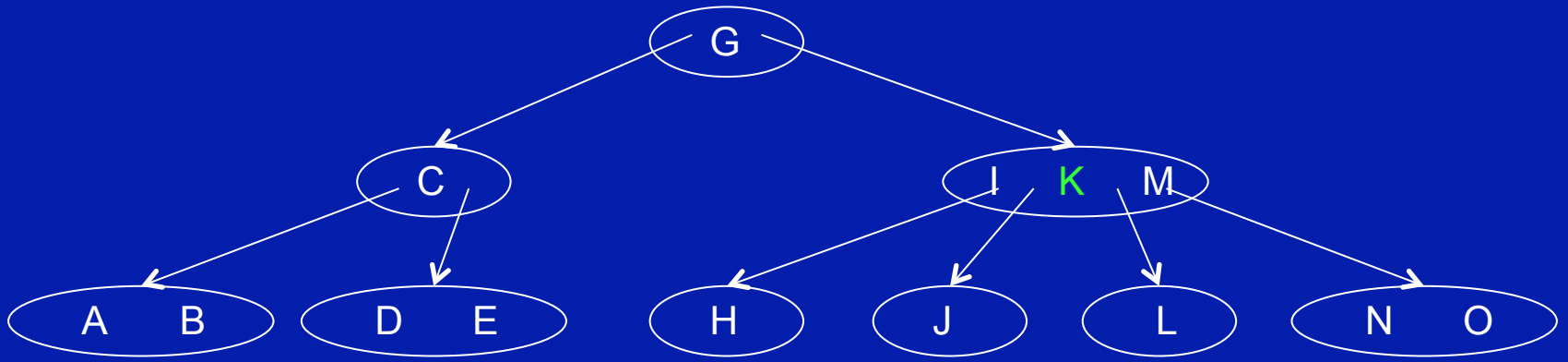
2-3 trees



To insert key L, again we search to find the node where it should be and add it.

But this node now has too many keys, so we split the node into two nodes and send the middle key up to the parent.

2-3 trees

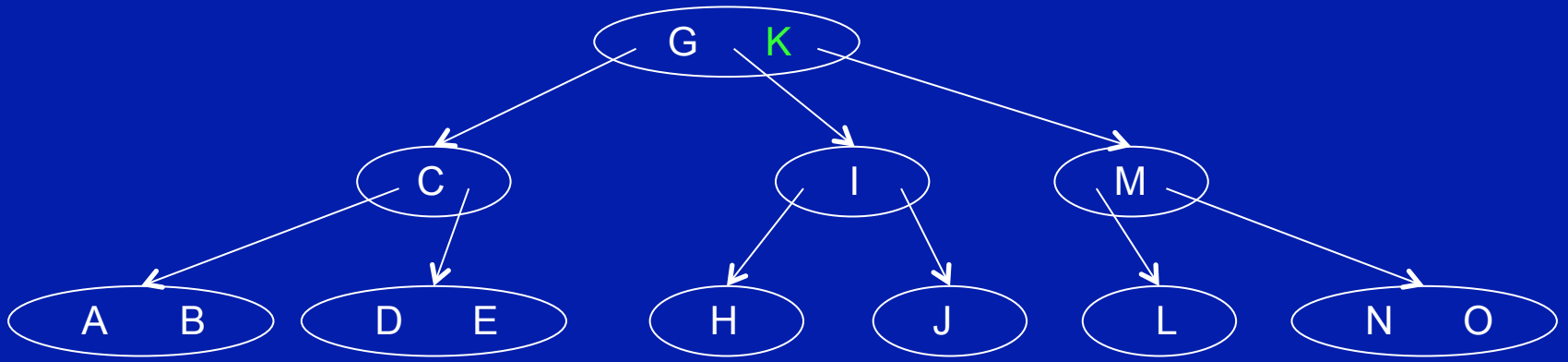


To insert key L, again we search to find the node where it should be and add it.

But this node now has too many keys, so we split the node into two nodes and send the middle key up to the parent.

And now this node has too many keys too, so we split it and send the middle key up again.

2-3 trees



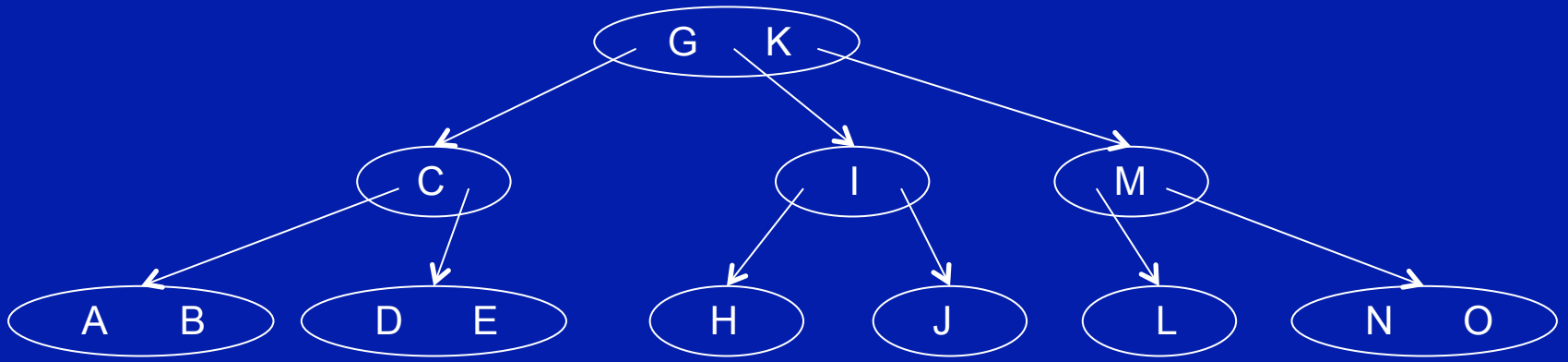
To insert key L, again we search to find the node where it should be and add it.

But this node now has too many keys, so we split the node into two nodes and send the middle key up to the parent.

And now this node has too many keys too, so we split it and send the middle key up again.

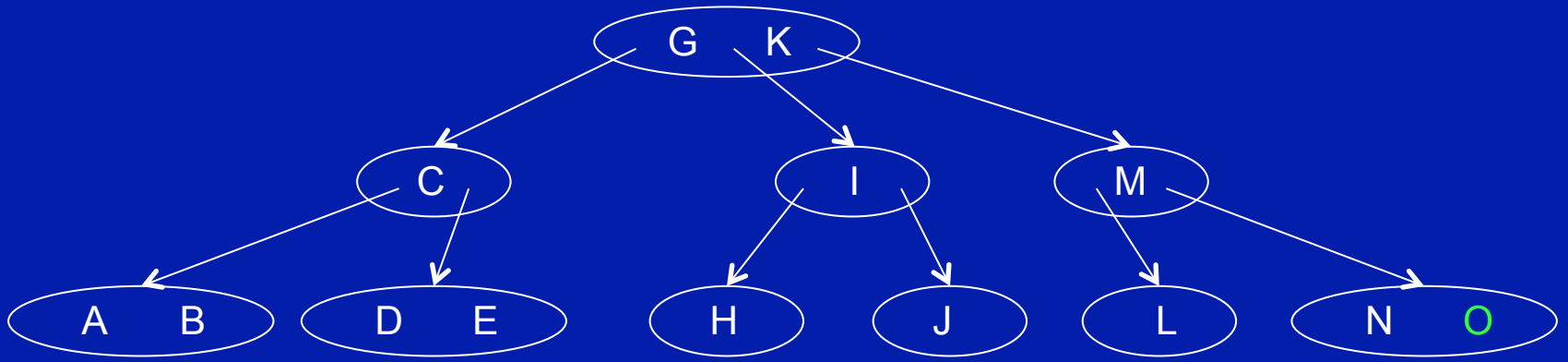
The root node is good, so we're done.

2-3 trees



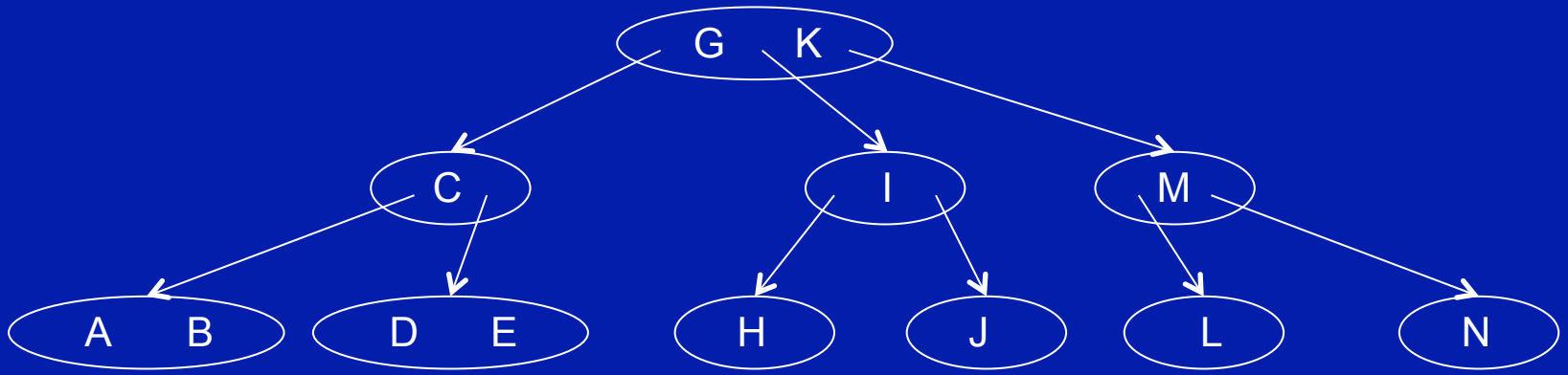
Deletion is sort of the inverse of insertion. If the item to be deleted is in a “leaf” node with two items, we just delete it.

2-3 trees



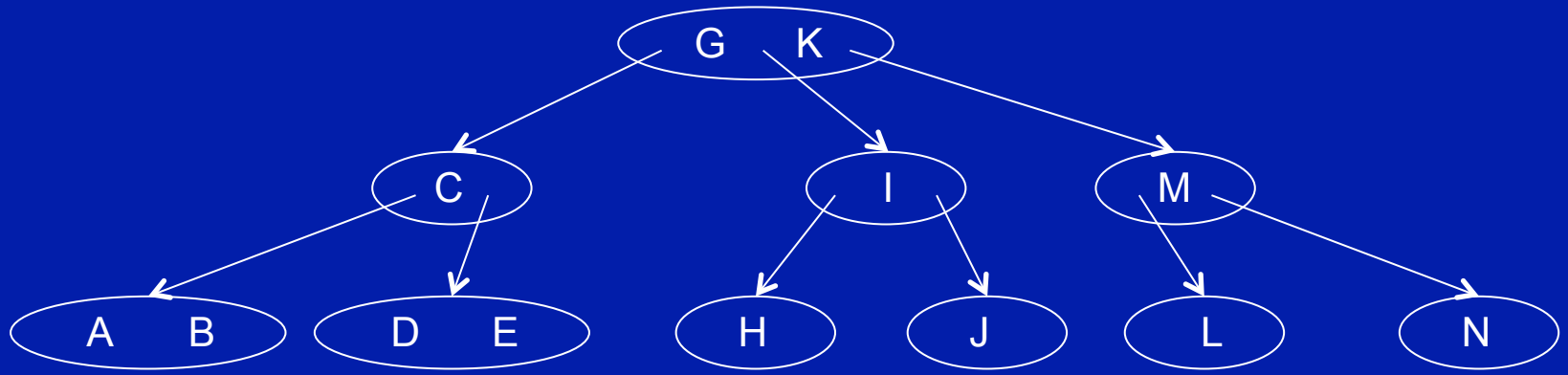
Deletion is sort of the inverse of insertion. If the item to be deleted is in a “leaf” node with two items, we just delete it.

2-3 trees



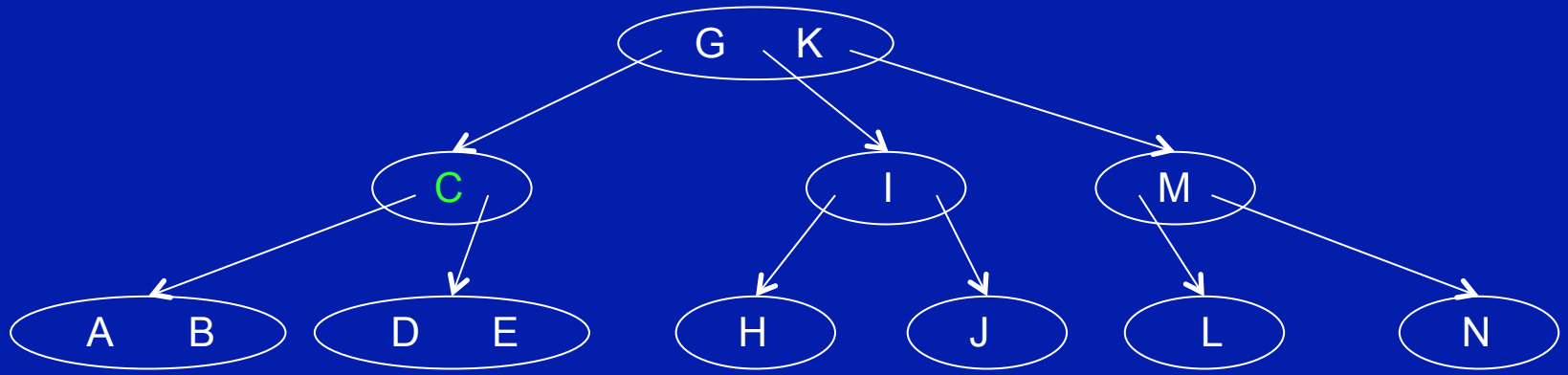
Deletion is sort of the inverse of insertion. If the item to be deleted is in a “leaf” node with two items, we just delete it.

2-3 trees



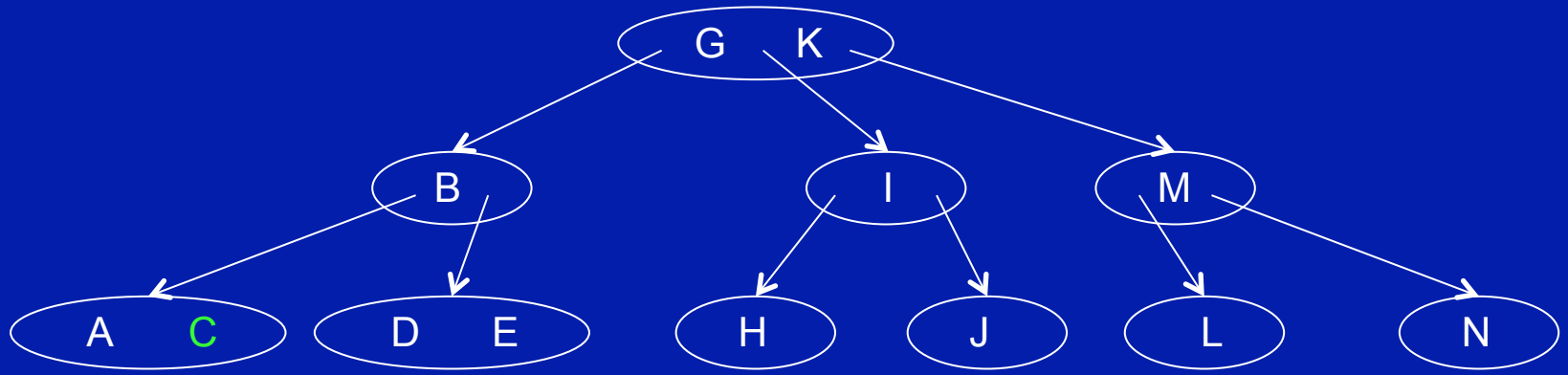
If the item is not in a “leaf”, it is swapped with its inorder predecessor from a “leaf” node and then deleted from the “leaf” node.

2-3 trees



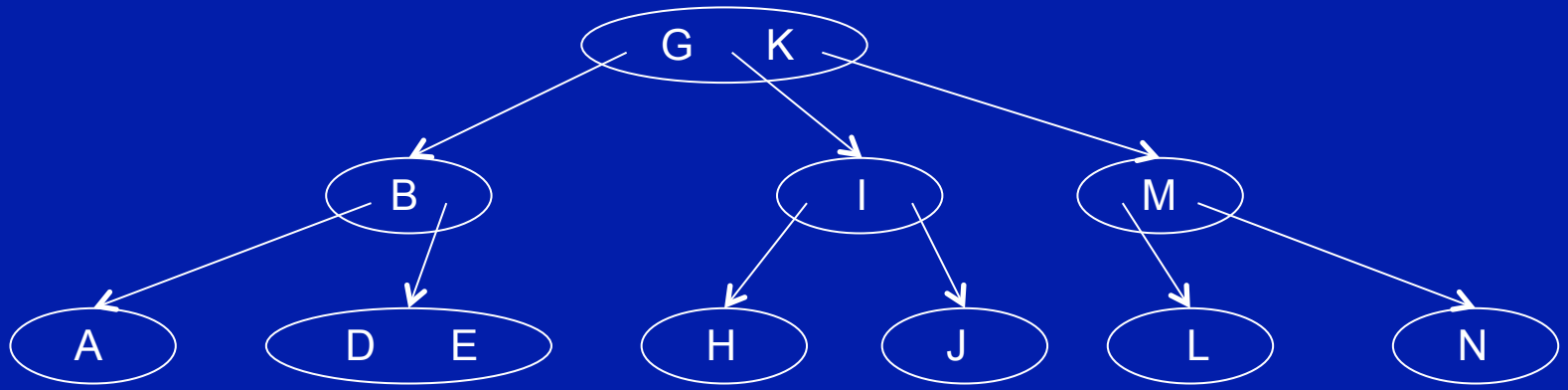
If the item is not in a “leaf”, it is swapped with its inorder predecessor from a “leaf” node and then deleted from the “leaf” node.

2-3 trees



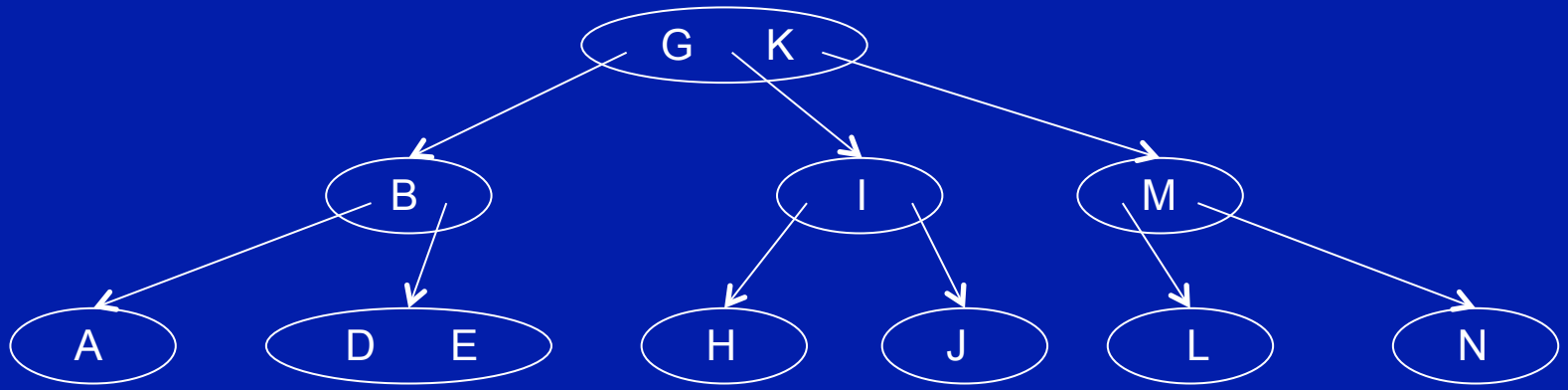
If the item is not in a “leaf”, it is swapped with its inorder predecessor from a “leaf” node and then deleted from the “leaf” node.

2-3 trees



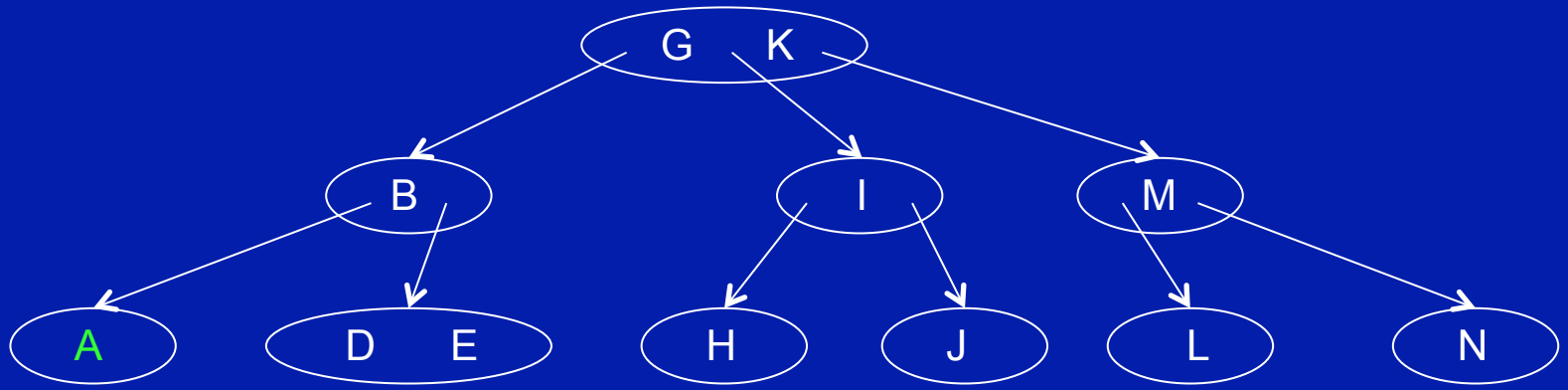
If the item is not in a “leaf”, it is swapped with its inorder predecessor from a “leaf” node and then deleted from the “leaf” node.

2-3 trees



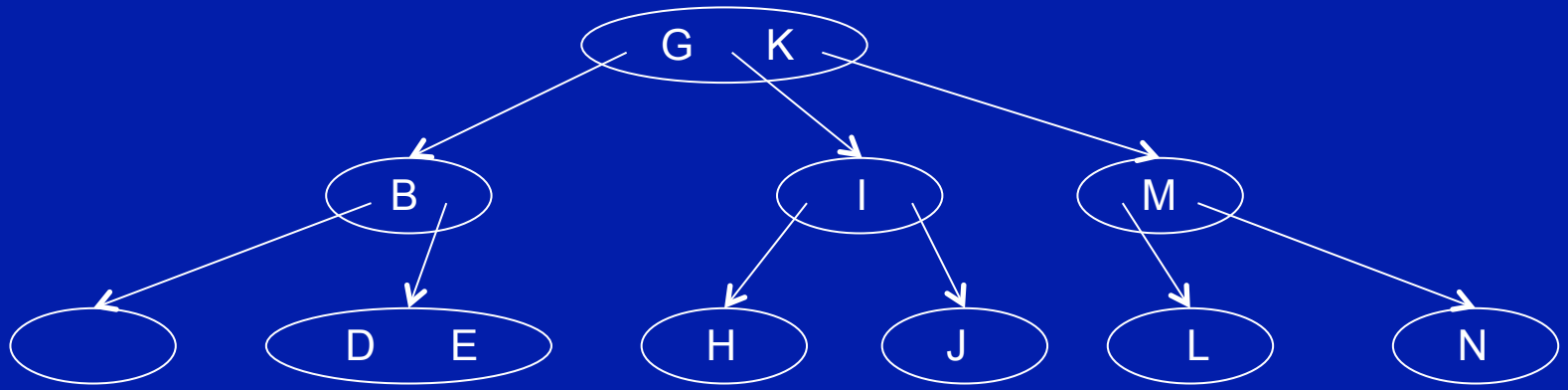
If removing a node from a leaf causes the leaf to become empty, items from the sibling and parent can be redistributed into that leaf.

2-3 trees



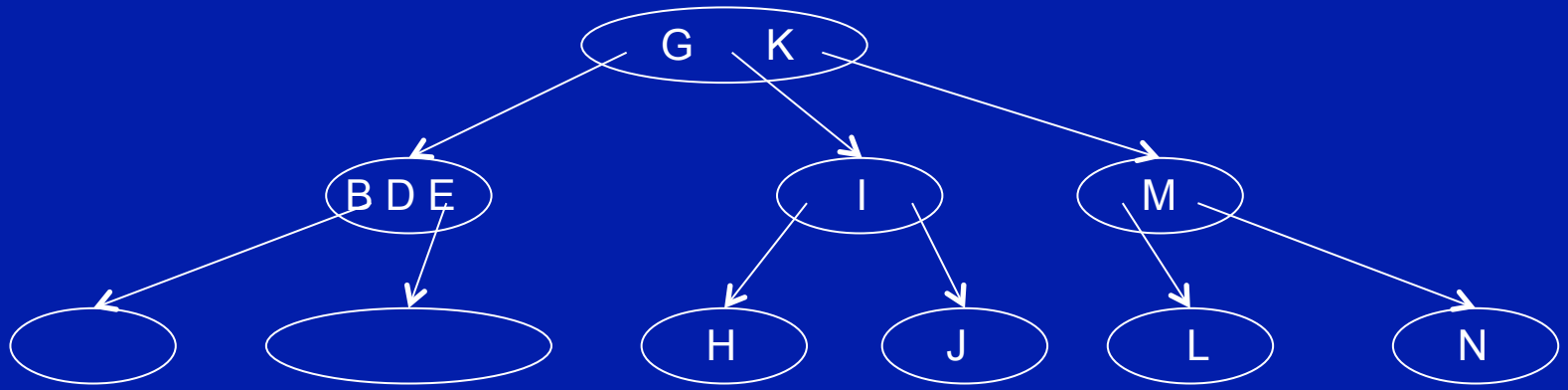
If removing a node from a leaf causes the leaf to become empty, items from the sibling and parent can be redistributed into that leaf.

2-3 trees



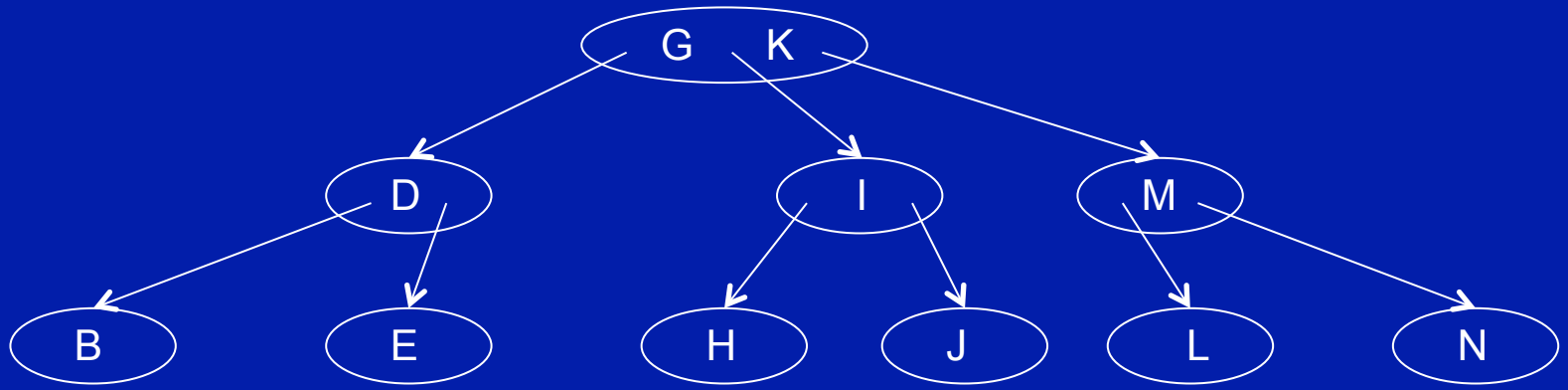
If removing a node from a leaf causes the leaf to become empty, items from the sibling and parent can be redistributed into that leaf.

2-3 trees



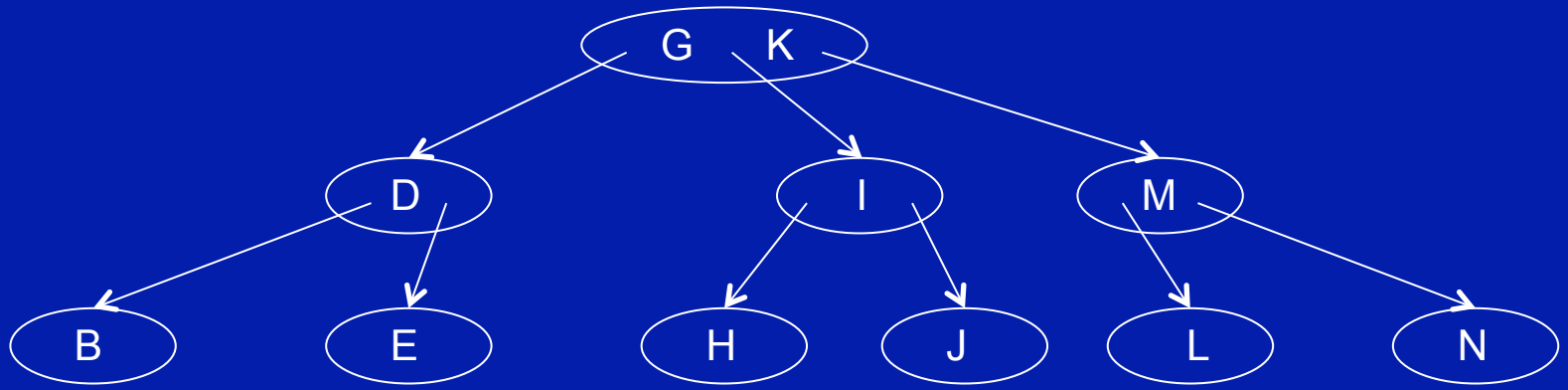
If removing a node from a leaf causes the leaf to become empty, items from the sibling and parent can be redistributed into that leaf.

2-3 trees



If removing a node from a leaf causes the leaf to become empty, items from the sibling and parent can be redistributed into that leaf.

2-3 trees



Your textbook's descriptions of deletion from B-trees aren't much more formal than what you've just seen. You should spend some time building up whatever intuition you can from the examples in the book.

2-3 tree performance

2-3 trees require fewer complicated manipulations than AVL trees. There are no rotations.

The number of items that a 2-3 tree can hold is between $2^h - 1$ (all 2-nodes) and $3^h - 1$ (all 3-nodes).

Thus the height of a 2-3 tree is between $\log_3 n$ and $\log_2 n$.

Consequently the search time is O(log n), since logarithms are all related by a constant factor and we ignore constants in the land of Big-O.

2-3-4 trees

A 2-3-4 tree is another specialization of the B-tree. It is a B-tree where m = 4, so the 4-node is added to the 2-node and the 3-node.

Your book says that the addition of this new data item simplifies the insertion logic.

You should definitely give the 2-3-4 tree description in the book some consideration and determine whether the insertion logic really is simplified. (In other words, just because we don't talk about it here doesn't mean I don't expect you to be familiar with it.)

Heaps

Say you're at the grocery store. You put a few items in your cart and head for checkout. There are a couple of other sparsely-laden carts ahead of you, but ahead of those is the guy whose cart has enough food for the next five Thanksgiving dinners. Should you and the other sparsely-laden carts go ahead of him?

Heaps

Say you're at the grocery store. You put a few items in your cart and head for checkout. There are a couple of other sparsely-laden carts ahead of you, but ahead of those is the guy whose cart has enough food for the next five Thanksgiving dinners. Should you and the other sparsely-laden carts go ahead of him?

You click 'send' on an email to your significant other to say you'll meet him/her in 15 minutes at your favourite pub. But just before you clicked that button, another person using the same email system clicked 'send' on an uncompressed file of all the digital video he's taken since 2007. Whose email should go first?

Heaps

Say you're at the grocery store. You put a few items in your cart and head for checkout. There are a couple of other sparsely-laden carts ahead of you, but ahead of those is the guy whose cart has enough food for the next five Thanksgiving dinners. Should you and the other sparsely-laden carts go ahead of him?

You click 'send' on an email to your significant other to say you'll meet him/her in 15 minutes at your favourite pub. But just before you clicked that button, another person using the same email system clicked 'send' on an uncompressed file of all the digital video he's taken since 2007. Whose email should go first?

You put your to-do list in the printer queue right after the student in the next office hit the 'print' button on her 957-page doctoral dissertation with annotated bibliography. Which one should get printed first?

Heaps

Problems like this can be solved easily by searching the queue for “little jobs” that can be done before the “big jobs”.

If that dissertation in the printer queue holds up the small print requests of 50 other people, you have 50 unhappy people. If those 50 jobs go first, then at most you have 1 unhappy person. And that person isn’t expecting speedy printing anyway, so she probably won’t notice.

If the jobs/requests are being fed into a traditional FIFO queue, you can extract the “small jobs” by performing a linear search on the data structure. As the queue gets bigger, it takes more time to search for the “small jobs” ... time that could be spent on processing the jobs.

If we want to prioritize the entries in the queue, we’ll want a faster way of getting at the “small jobs”...

Heaps

A heap is a complete* binary tree in which

- If A is a parent node of B, then the value (or key) at A is ordered with respect to the value (or key) of B, and the same ordering applies throughout the heap. In other words, every subtree of a heap is a heap.

*Danger! Terminology confusion ahead.

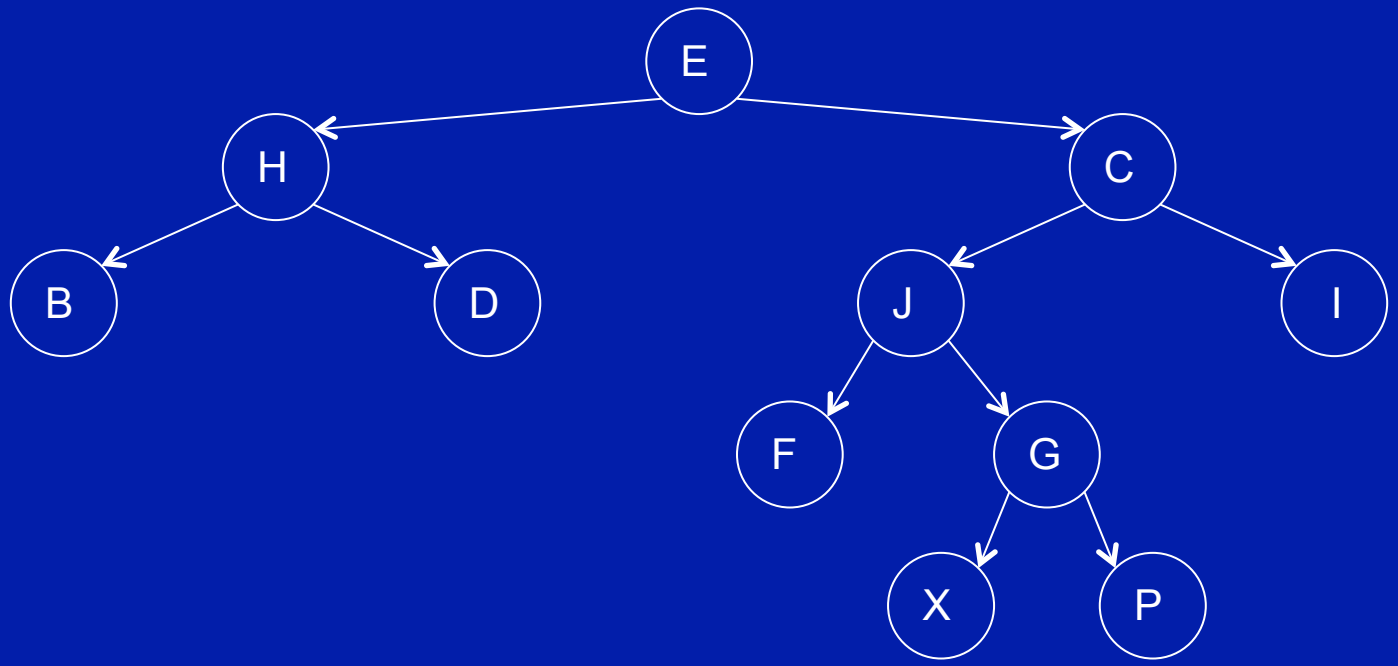
Tree terminology

A full binary tree is a binary tree in which each node has exactly 0 or 2 children. Each internal node has exactly 2 children, and each leaf has 0 children.

A complete binary tree is a binary tree of height h in which leaves appear only at two adjacent levels, depth h – 1 and depth h, and the leaves at the very bottom (depth h) are in the leftmost positions.

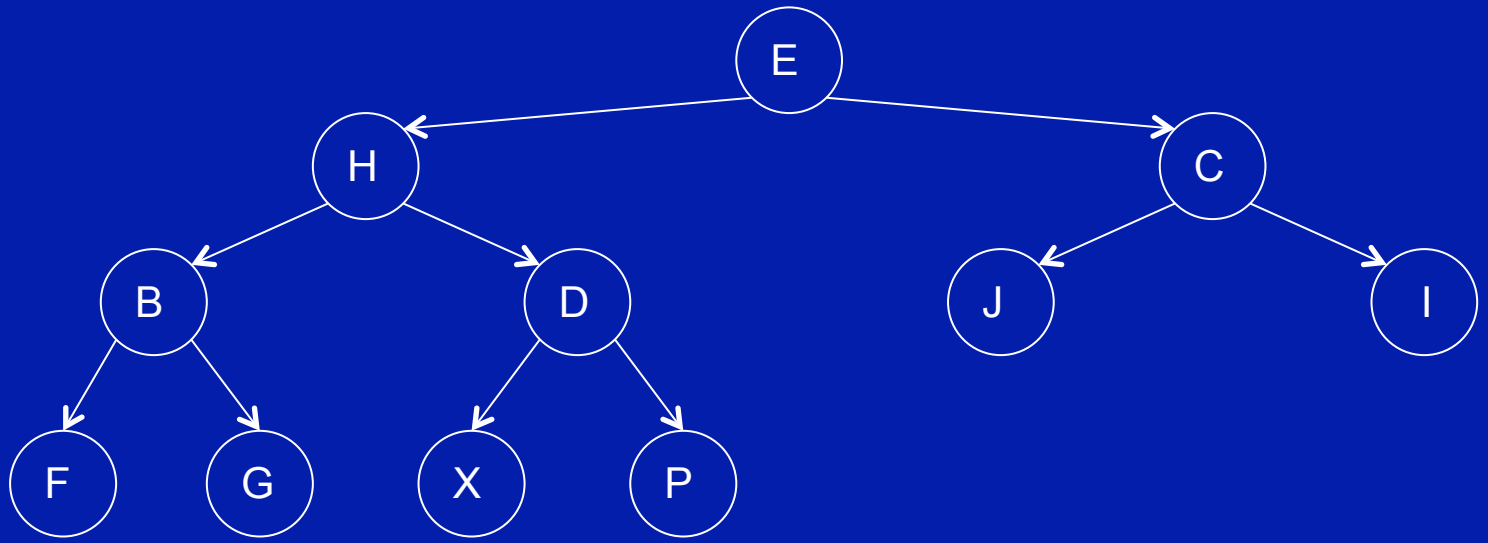
Some sources (e.g., Wikipedia) say that a binary tree is almost complete or nearly complete if the bottom level is not completely filled, and the tree is only complete if the bottom level is completely filled.

Tree terminology



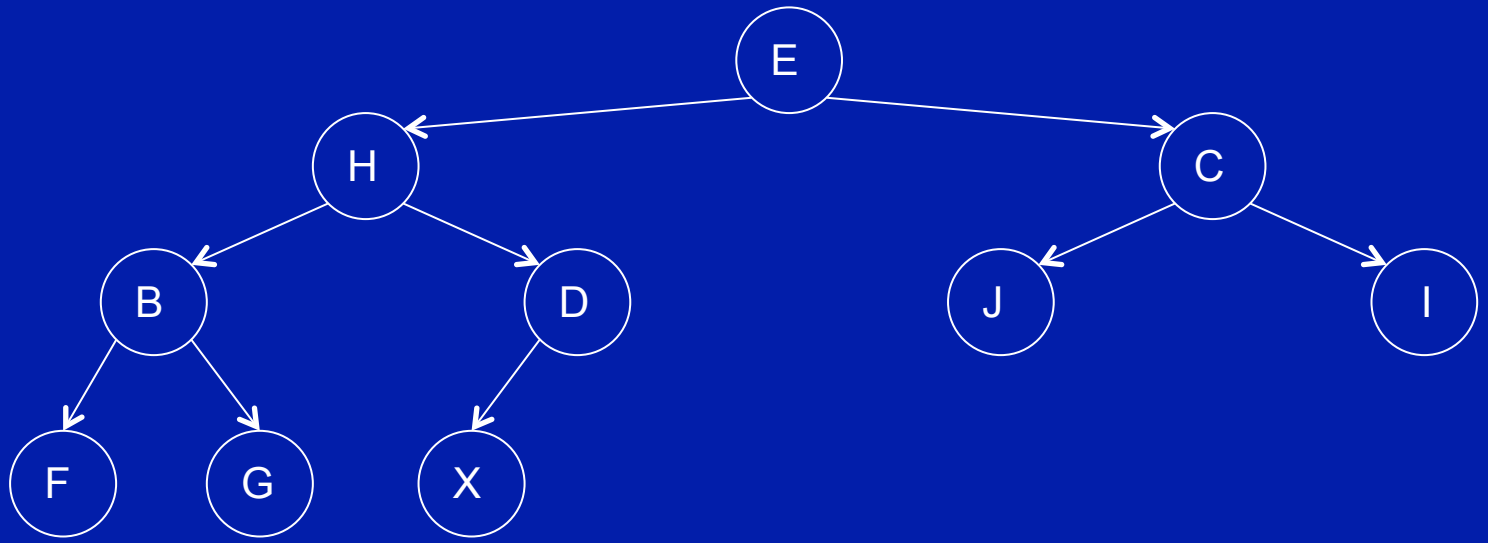
Complete according to your textbook?
Complete by other sources? No
Nearly complete? No

Tree terminology



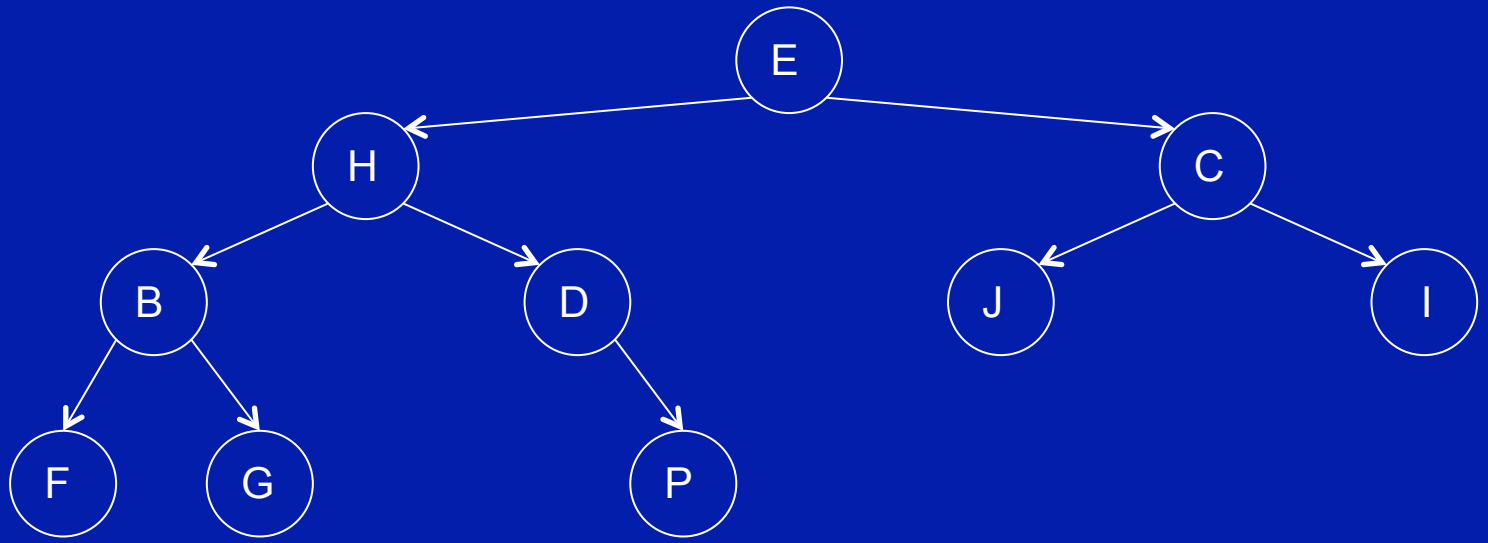
Complete according to your textbook? Yes
Complete by other sources? No
Nearly complete? Yes

Tree terminology



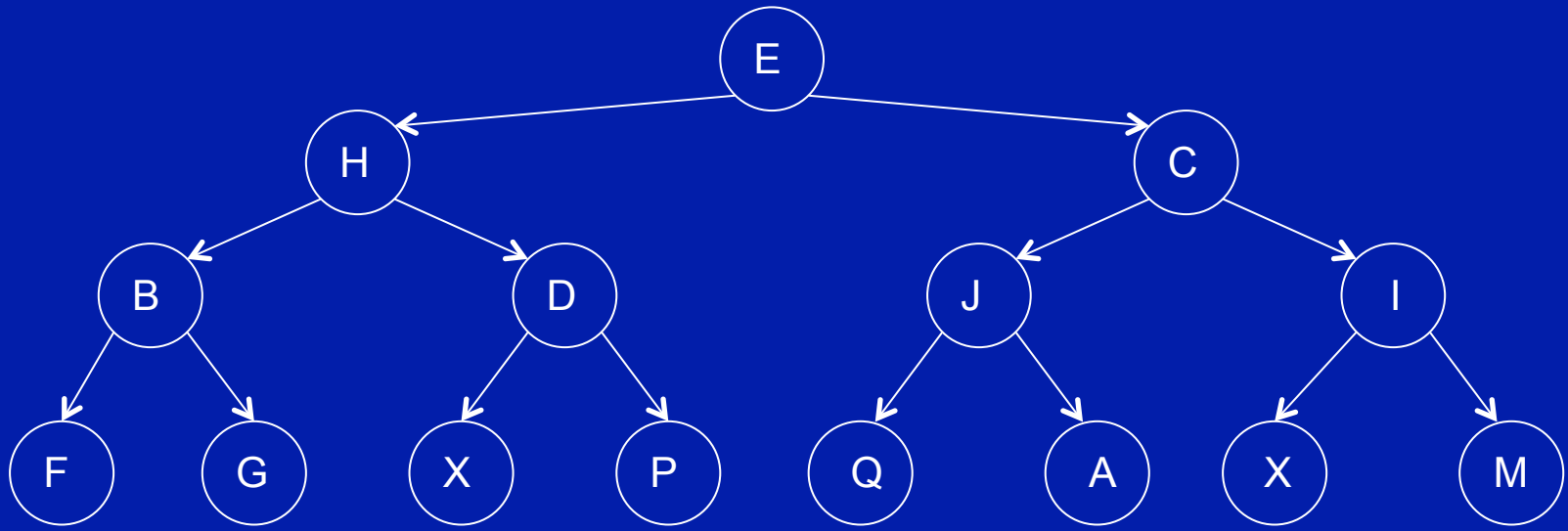
Complete according to your textbook? Yes
Complete by other sources? No
Nearly complete? Yes

Tree terminology



Complete according to your textbook? No
Complete by other sources? No
Nearly complete? No

Tree terminology



Complete according to your textbook? Yes
Complete by other sources? Yes
Nearly complete? No

Heaps

A heap is a complete (or nearly complete) binary tree in which

- If A is a parent node of B, then the value (or key) at A is ordered with respect to the value (or key) of B, and the same ordering applies throughout the heap. In other words, every subtree of a heap is a heap.

Heaps

If the values (or keys) of parent nodes are always greater than or equal to those of the child nodes and the largest value (or key) is at the root node, we have a maximum binary heap or *max heap*.

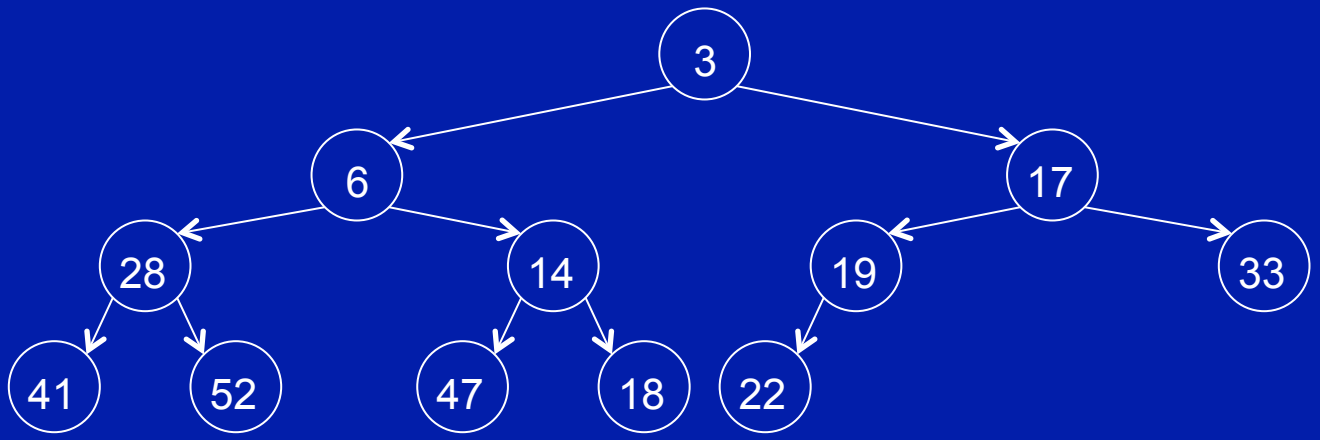
If the values (or keys) of parent nodes are always less than or equal to those of the child nodes and the smallest value (or key) is at the root node, we have a minimum binary heap or *min heap*.

All the examples in your textbook seem to be max heaps. So in lecture, we'll use min heaps just to give you some variety.

Also note the following: This type of heap is completely unrelated to the common pool of free memory from which dynamically allocated memory is assigned, which is also called a heap. Sorry for any confusion.

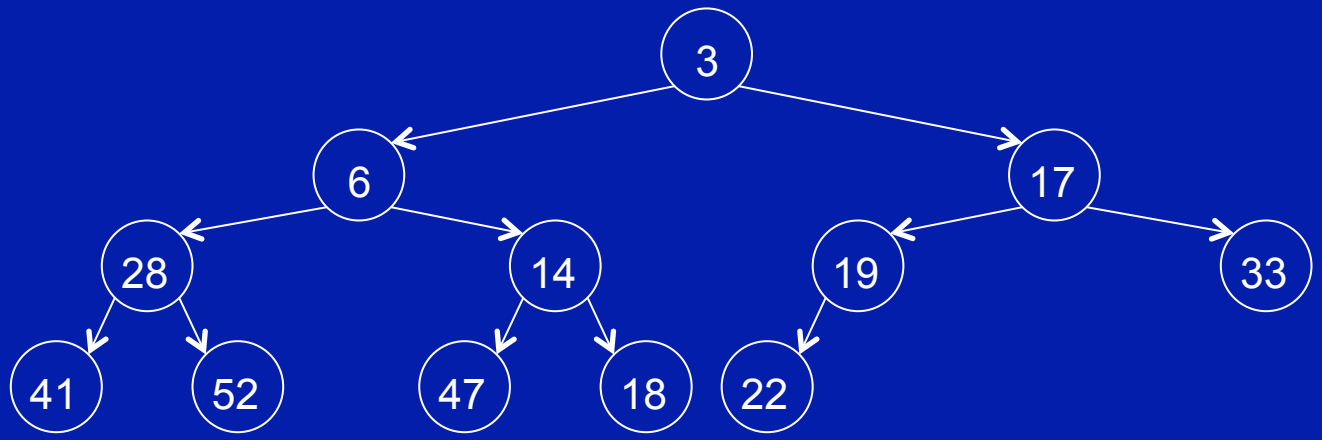
Heaps

Here is a min heap



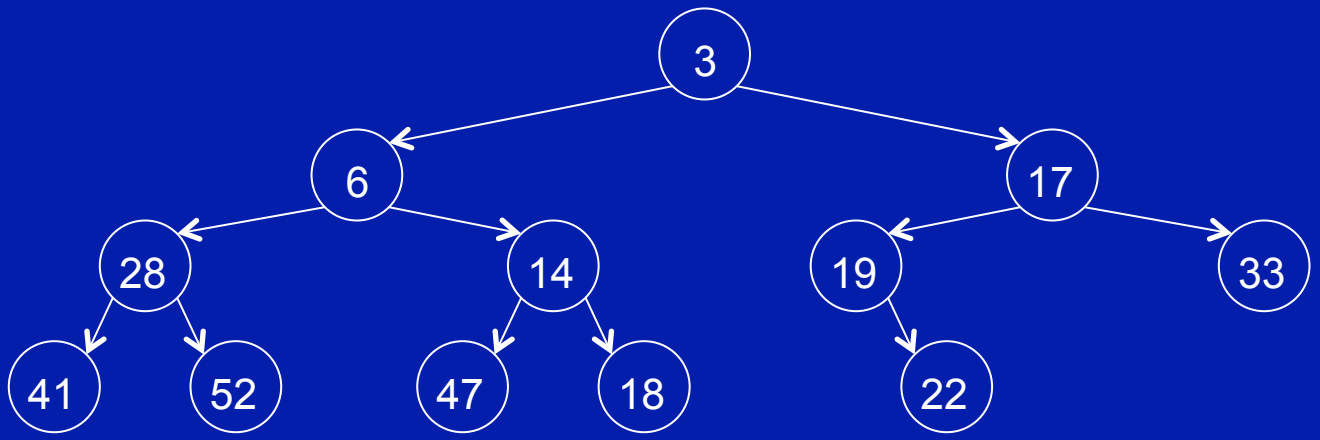
Heaps

Note that there is a top-to-bottom ordering, but there is no side-to-side ordering as we would see in a binary search tree.



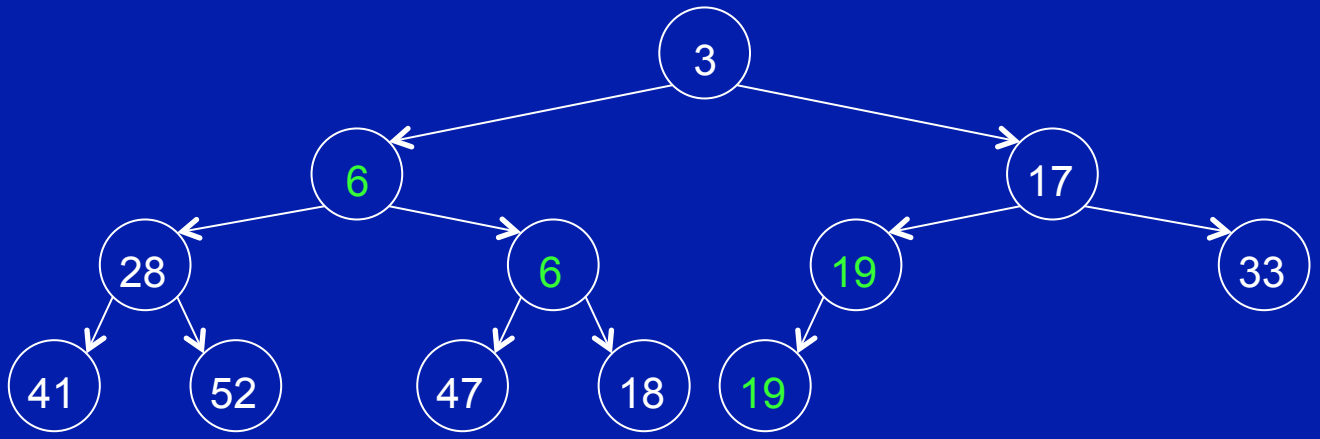
Heaps

This is not a heap



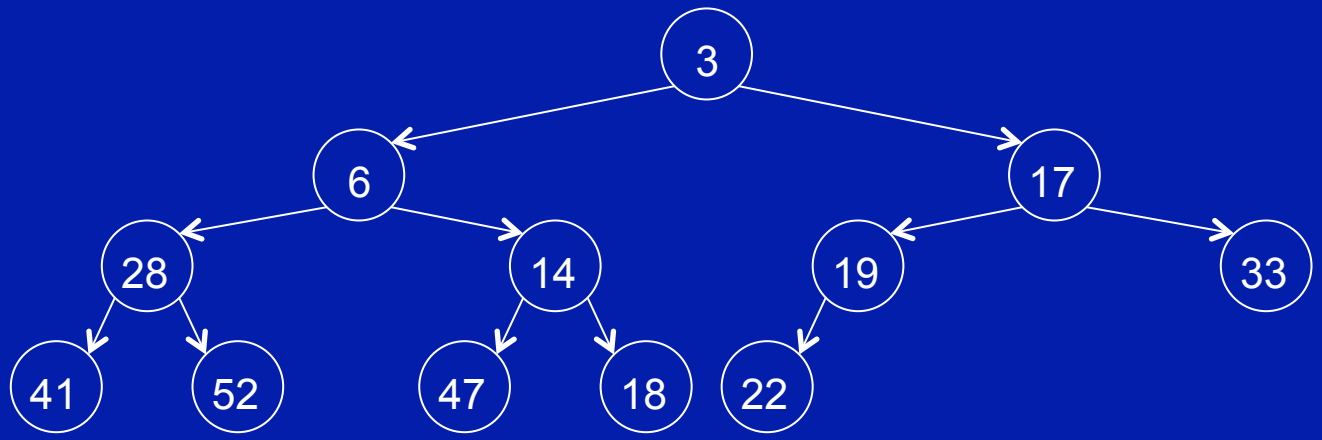
Heaps

A heap may contain the same values in different locations



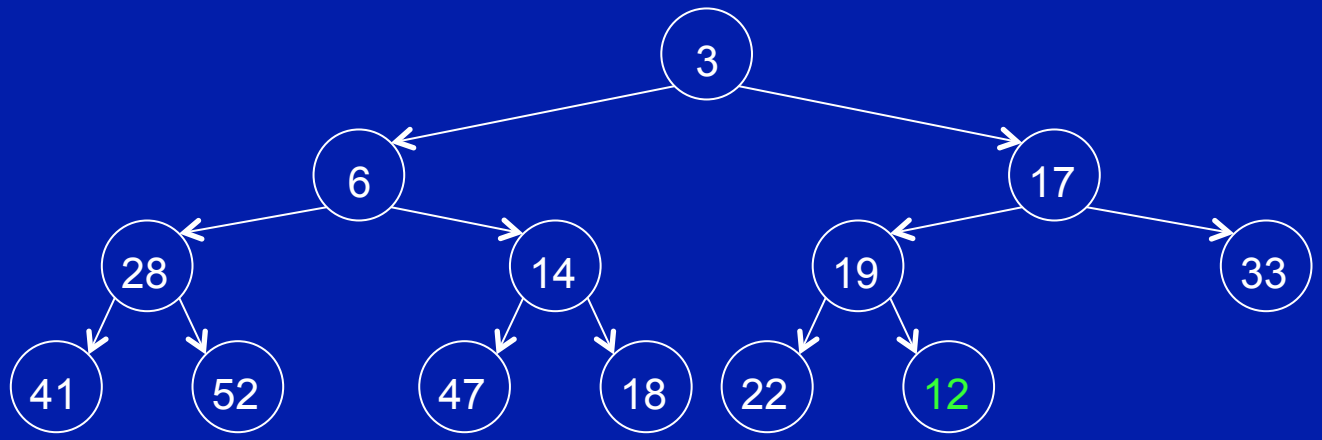
Insertion in a heap

Inserting a new item into a min heap starts with adding the item to the heap at the next available location in the complete binary tree:



Insertion in a heap

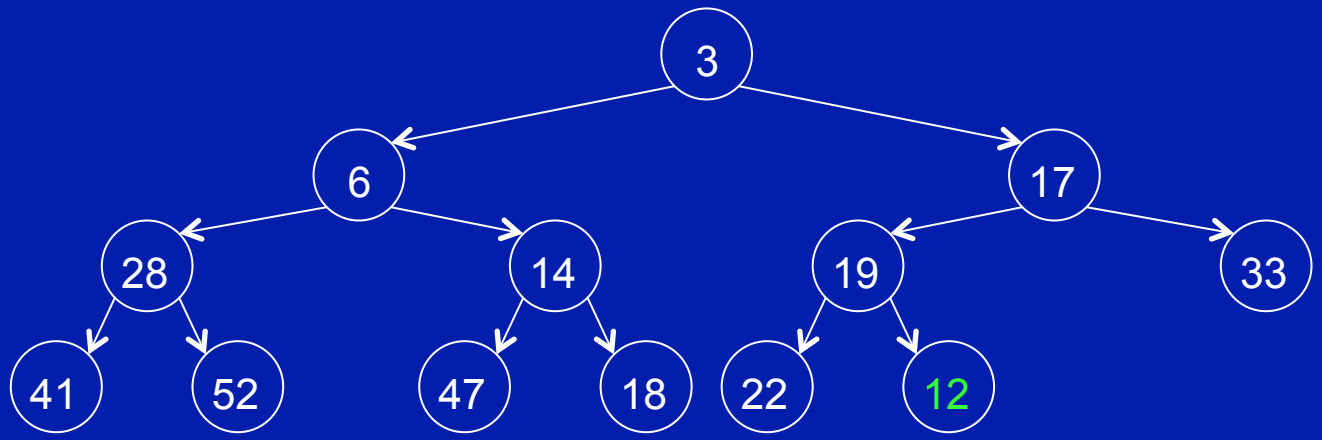
Inserting a new item into a min heap starts with adding the item to the heap at the next available location in the complete binary tree:



Insertion in a heap

Algorithm for insertion into a min heap:

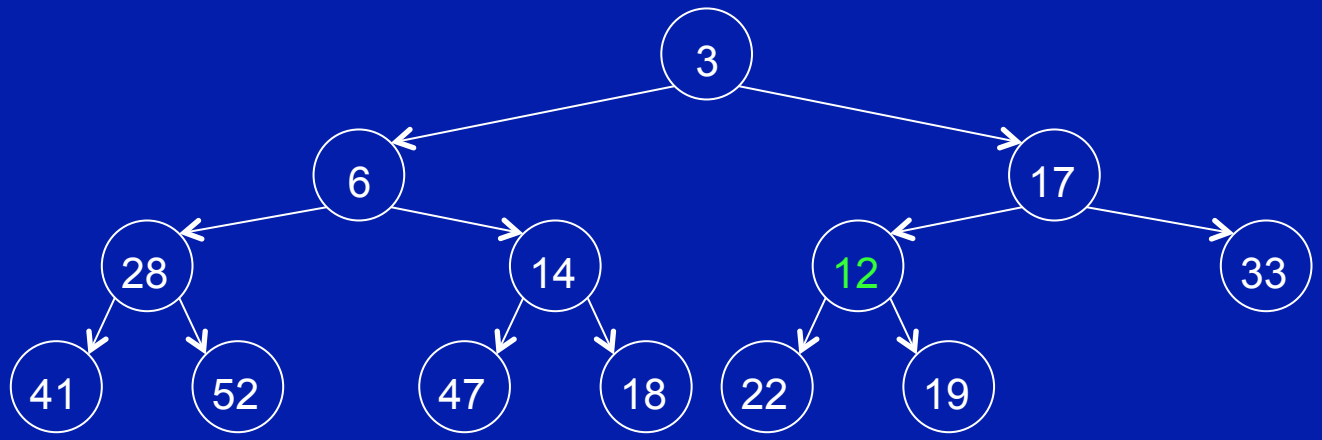
1. insert the new item in the next position at the bottom of the heap
2. while new item is not at the root and new item is smaller than its parent
3. swap the new item with its parent, moving the new item up the heap



Insertion in a heap

Algorithm for insertion into a min heap:

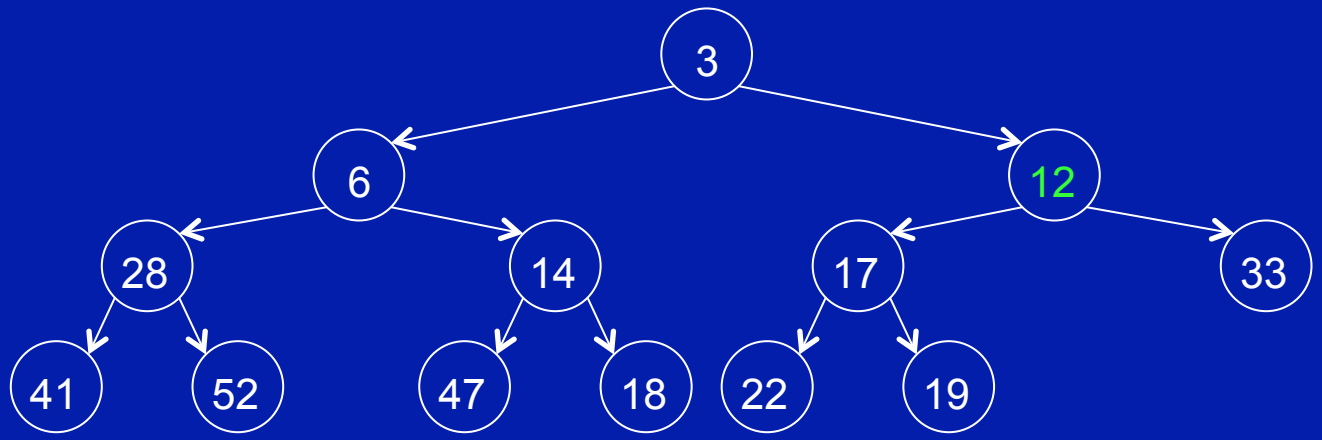
1. insert the new item in the next position at the bottom of the heap
2. while new item is not at the root and new item is smaller than its parent
3. swap the new item with its parent, moving the new item up the heap



Insertion in a heap

Algorithm for insertion into a min heap:

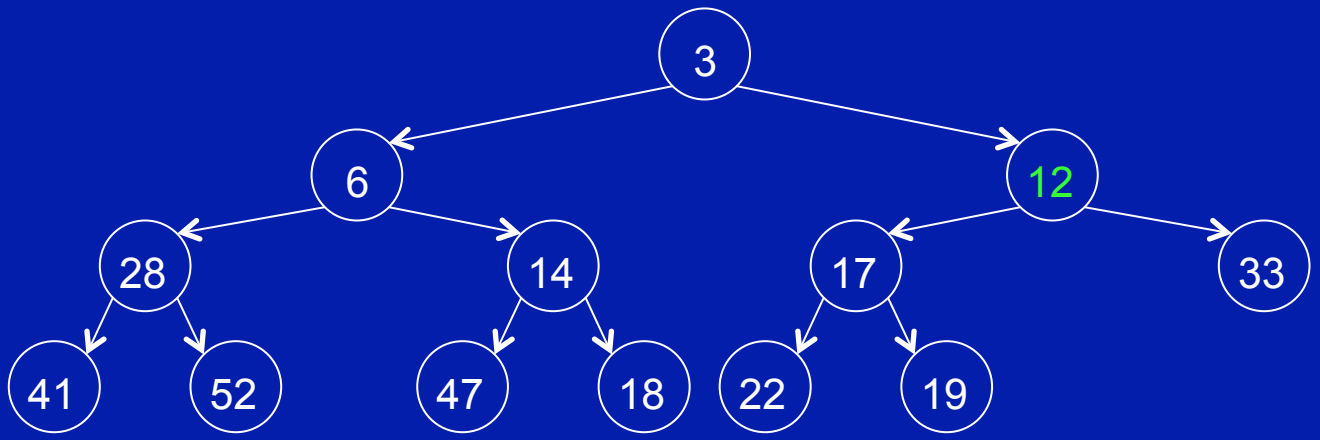
1. insert the new item in the next position at the bottom of the heap
2. while new item is not at the root and new item is smaller than its parent
3. swap the new item with its parent, moving the new item up the heap



Insertion in a heap

Some books call this part of the algorithm ReheapUp or Sift-Up

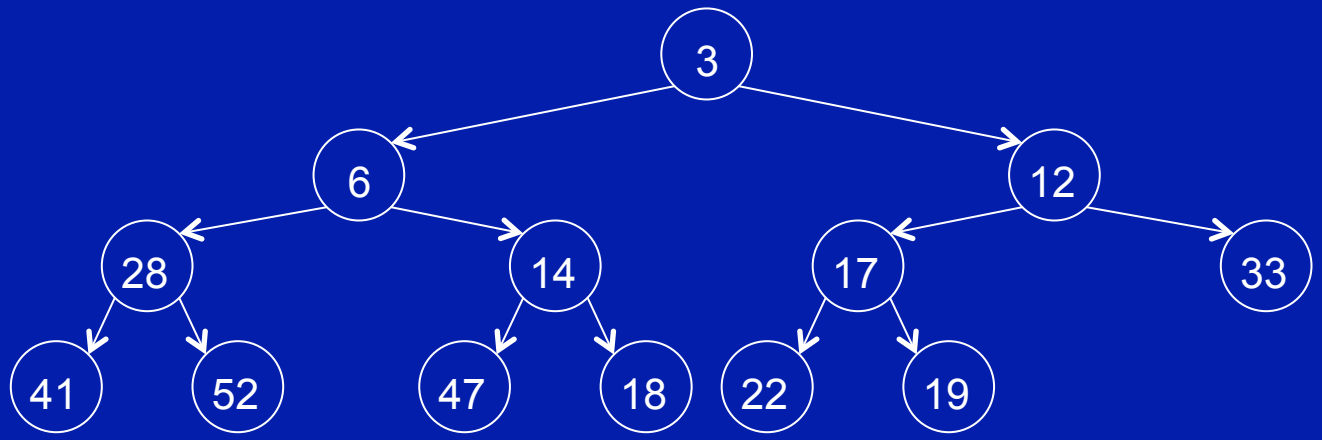
1. insert the new item in the next position at the bottom of the heap
2. while new item is not at the root and new item is smaller than its parent
3. swap the new item with its parent, moving the new item up the heap



Deletion from a heap

Algorithm for deletion from a min heap:

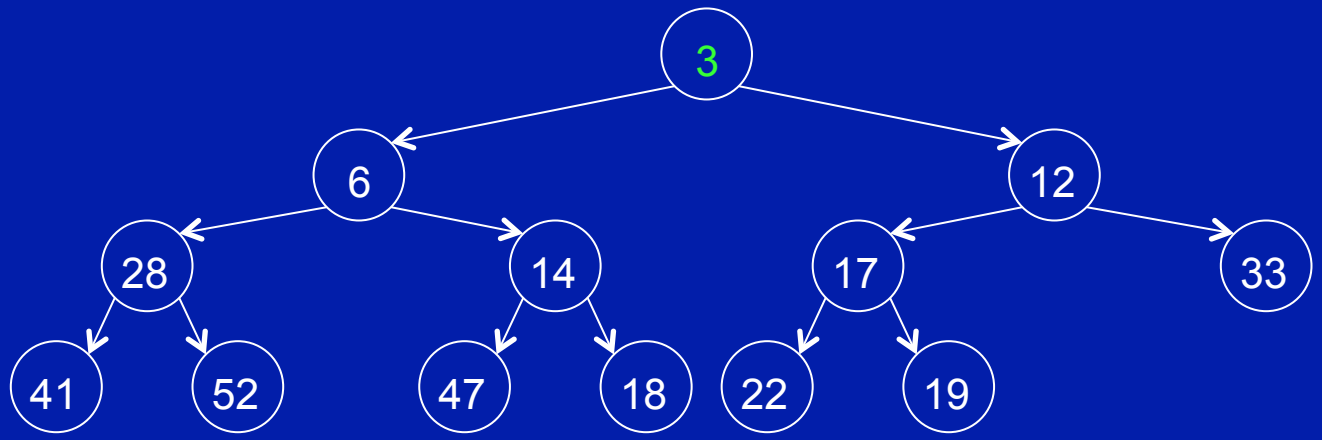
1. replace the item at the root node with the last item in the heap (LIH)
2. while item LIH has children and item LIH is larger than at least one child
3. swap item LIH with the smaller of its children, moving LIH down the heap



Deletion from a heap

Algorithm for deletion from a min heap:

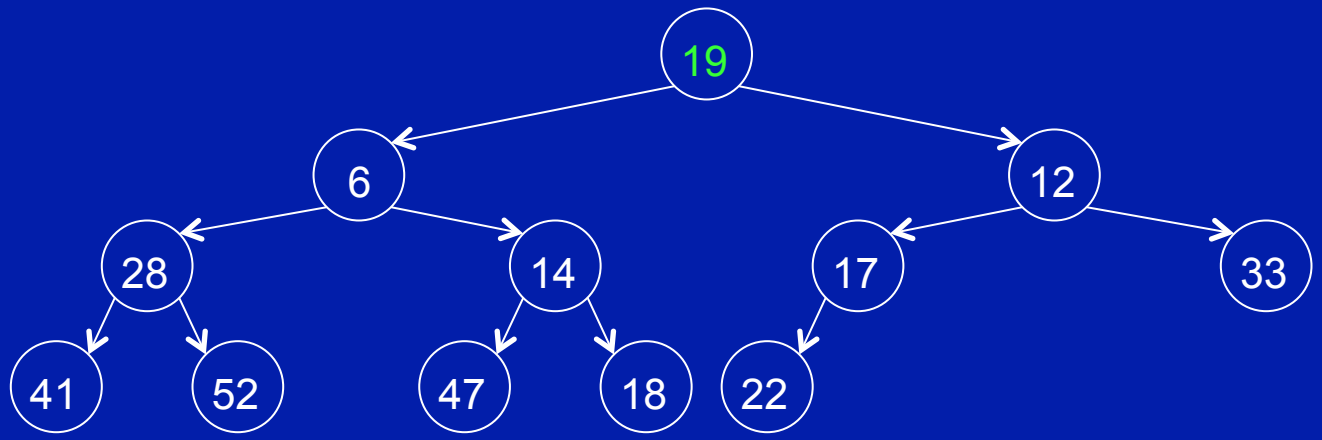
1. replace the item at the root node with the last item in the heap (LIH)
2. while item LIH has children and item LIH is larger than at least one child
3. swap item LIH with the smaller of its children, moving LIH down the heap



Deletion from a heap

Algorithm for deletion from a min heap:

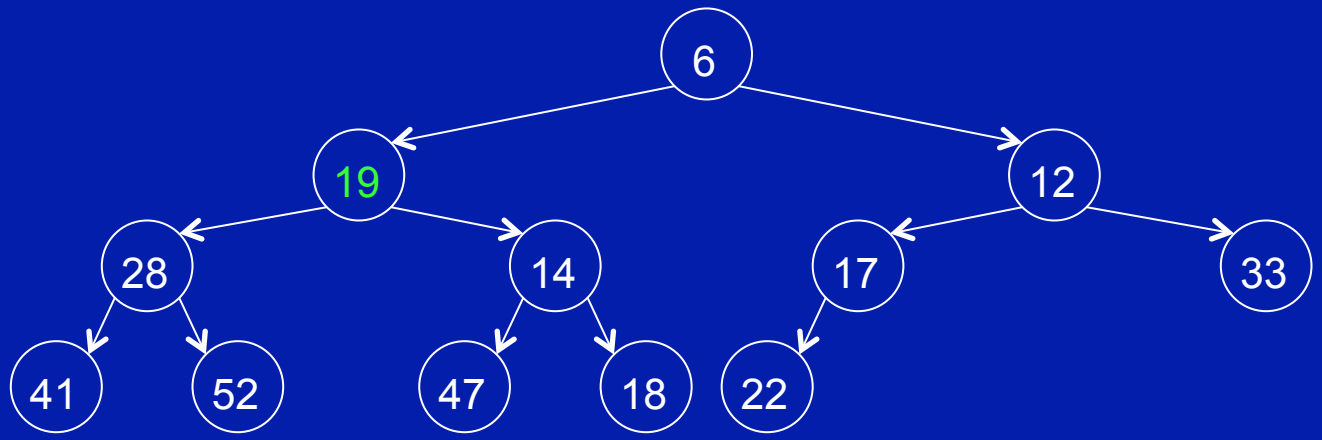
1. replace the item at the root node with the last item in the heap (LIH)
2. while item LIH has children and item LIH is larger than at least one child
3. swap item LIH with the smaller of its children, moving LIH down the heap



Deletion from a heap

Algorithm for deletion from a min heap:

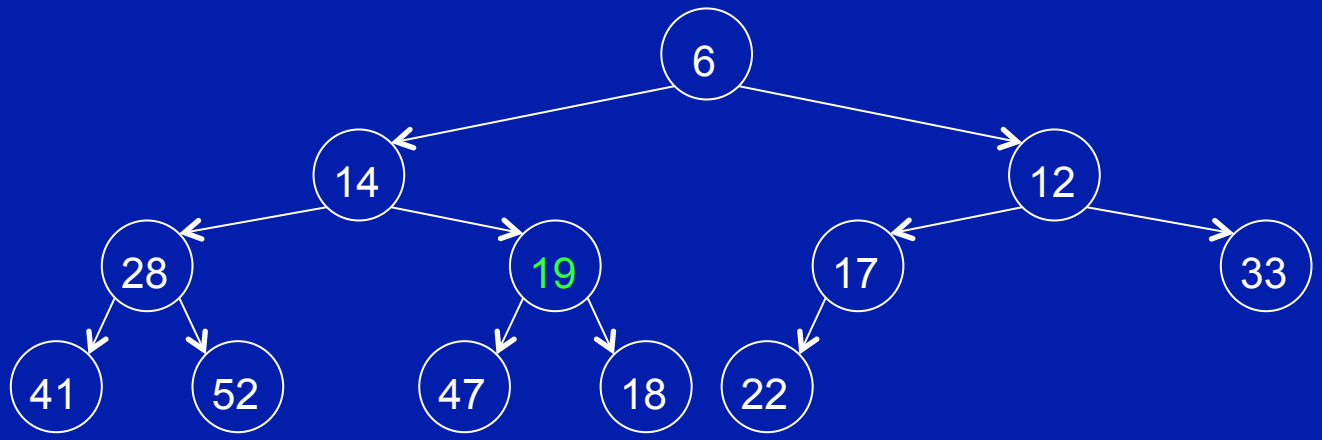
1. replace the item at the root node with the last item in the heap (LIH)
2. while item LIH has children and item LIH is larger than at least one child
3. swap item LIH with the smaller of its children, moving LIH down the heap



Deletion from a heap

Algorithm for deletion from a min heap:

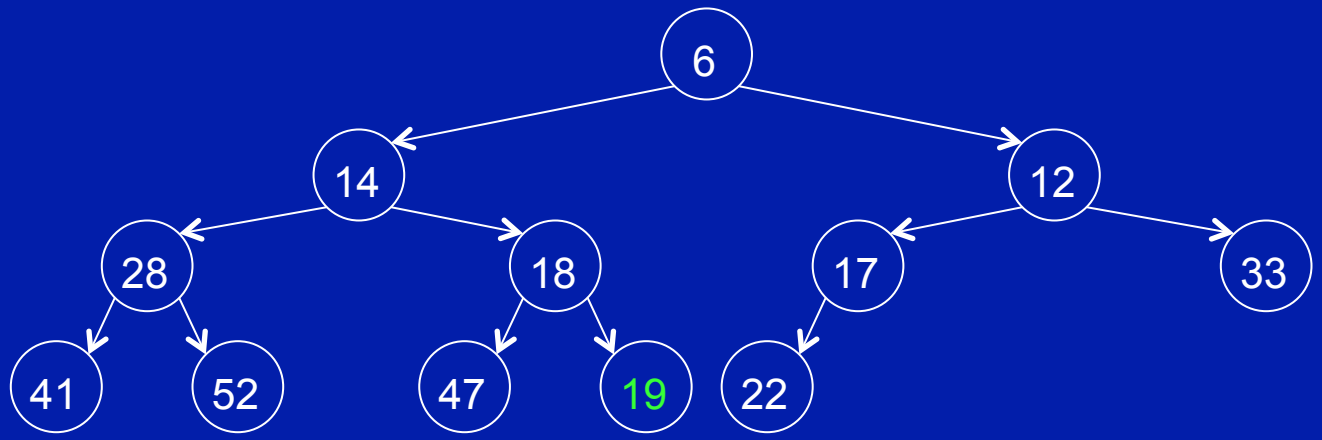
1. replace the item at the root node with the last item in the heap (LIH)
2. while item LIH has children and item LIH is larger than at least one child
3. swap item LIH with the smaller of its children, moving LIH down the heap



Deletion from a heap

Algorithm for deletion from a min heap:

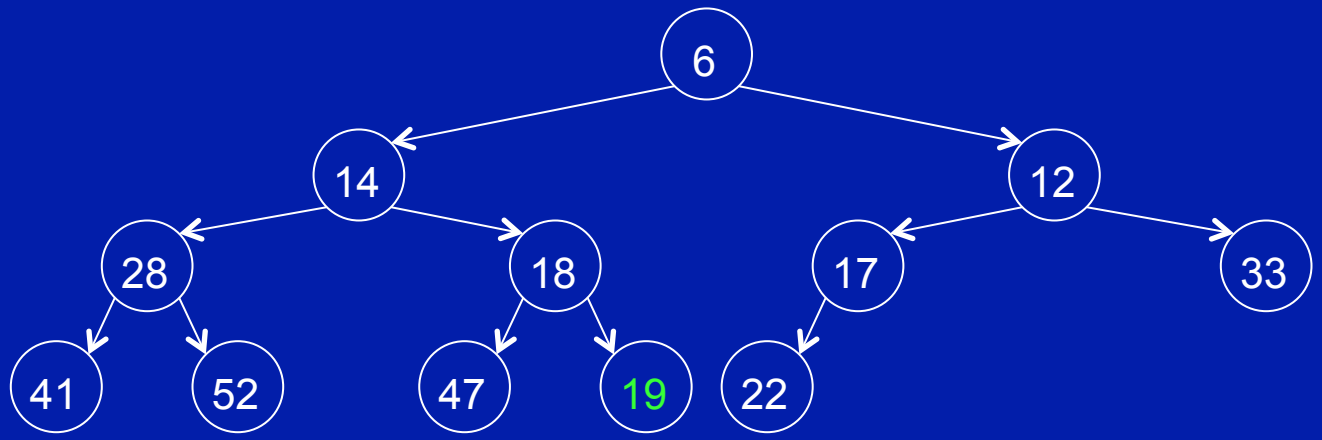
1. replace the item at the root node with the last item in the heap (LIH)
2. while item LIH has children and item LIH is larger than at least one child
3. swap item LIH with the smaller of its children, moving LIH down the heap



Deletion from a heap

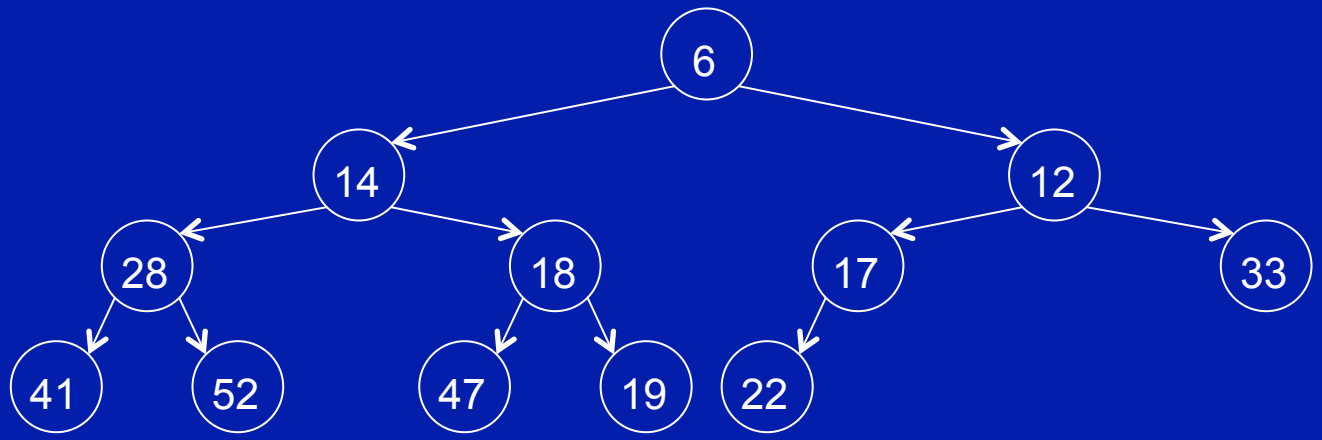
Some books call this part of the algorithm ReheapDown or Sift-Down

1. replace the item at the root node with the last item in the heap (LIH)
2. while item LIH has children and item LIH is larger than at least one child
3. swap item LIH with the smaller of its children, moving LIH down the heap



Deletion from a heap

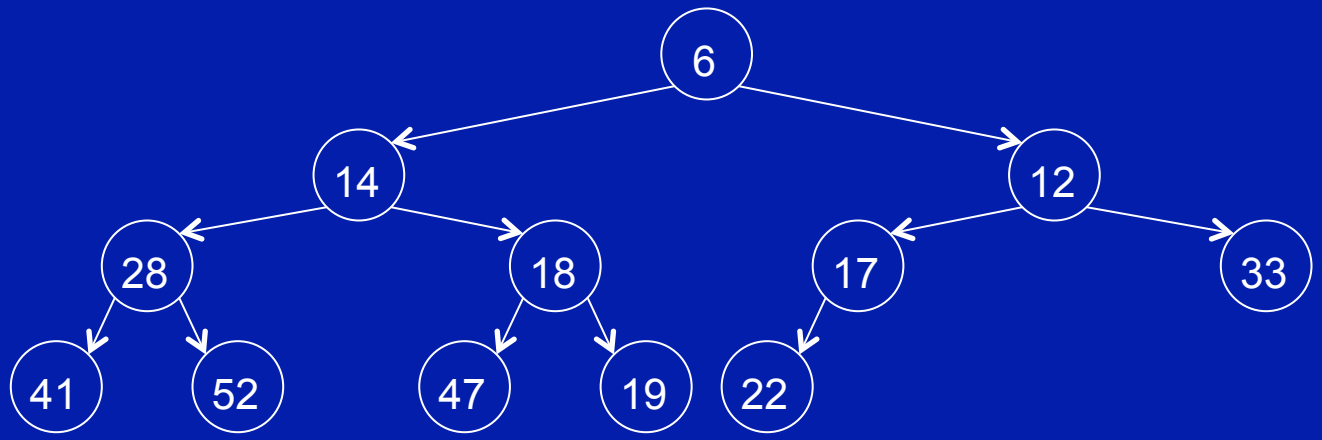
Usually, deletion is from the root. But you can delete from elsewhere in the heap. To do so, you replace the deleted node with LIH. Then reheap LIH up, then reheap it down. You get to practice that in today's exercises.



Time complexity

When doing a ReheapUp or ReheapDown, the number of operations depends on the height of the tree.

We only ever traverse one path of the tree.

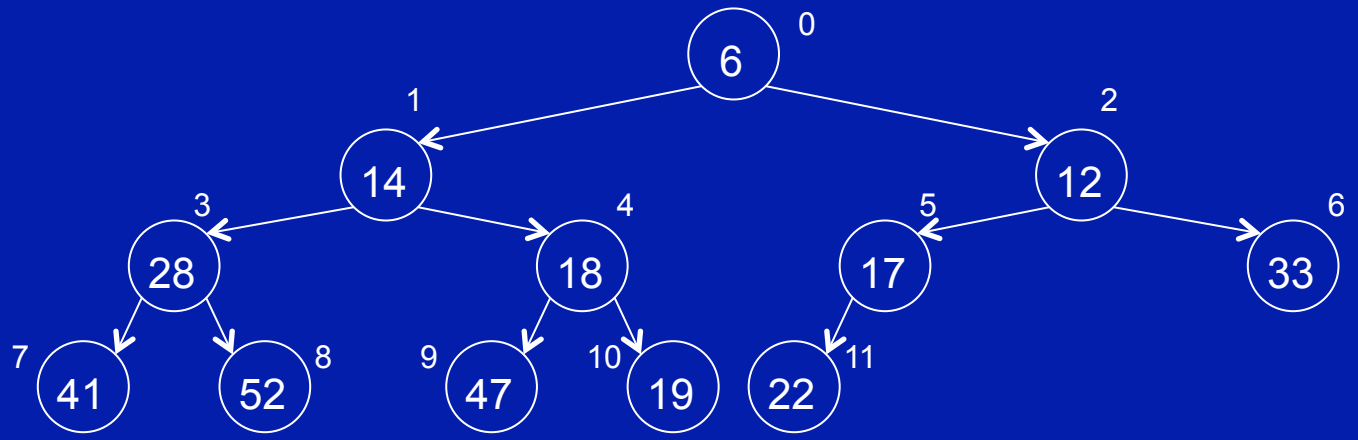


A complete binary tree of height h always has between 2^h and $2^{h+1} – 1$ nodes (= n). The height of the heap is, therefore, floor(lg n).

So the time complexity of ReheapUp and ReheapDown is O(lg n).

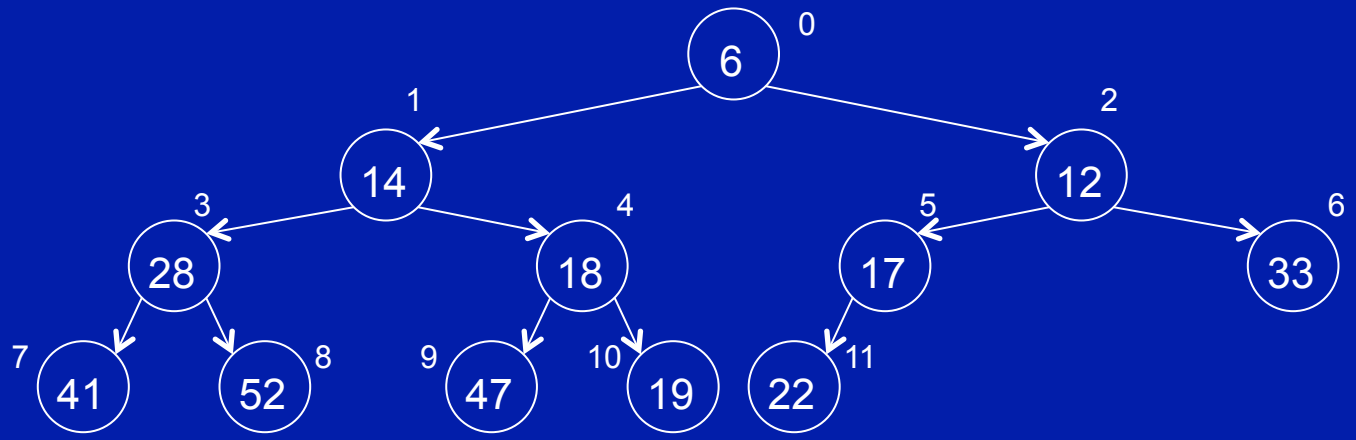
Implementing a heap

If we number the nodes in a complete binary tree by level, and left to right within a level, we get:



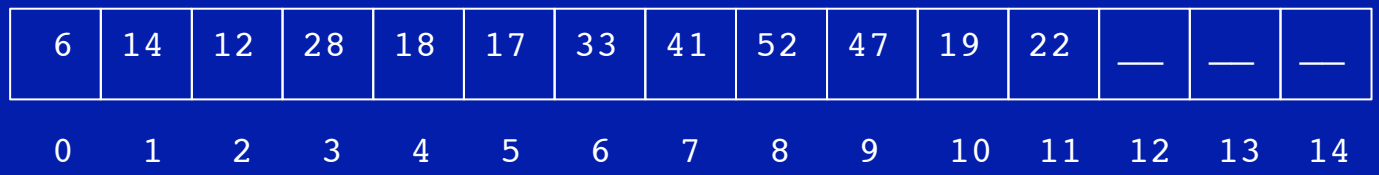
Implementing a heap

If we number the nodes in a complete binary tree by level, and left to right within a level, we get:



While other types of trees are implemented as linked list structures, a complete binary tree can be represented in contiguous storage (e.g., array, vector) using the numbering or indexing described above...

Implementing a heap



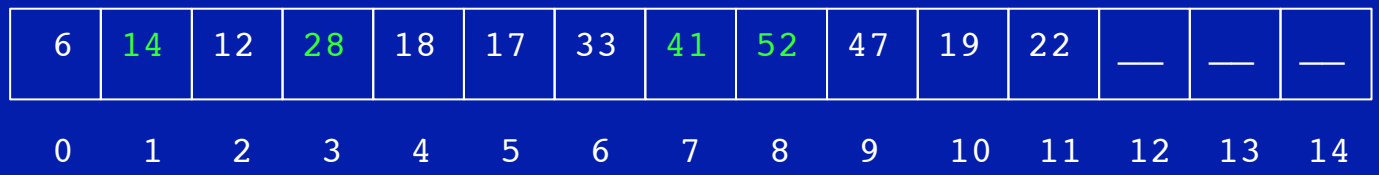
Now we can navigate the heap with simple index arithmetic instead of link traversals. Given a node at index k,

we can find the left child of k at index: $2k + 1$

we can find the right child of k at index: $2k + 2$

we can find the parent of k at index: $\lfloor (k - 1) / 2 \rfloor$

Implementing a heap



Now we can navigate the heap with simple index arithmetic instead of link traversals. Given a node at index k,

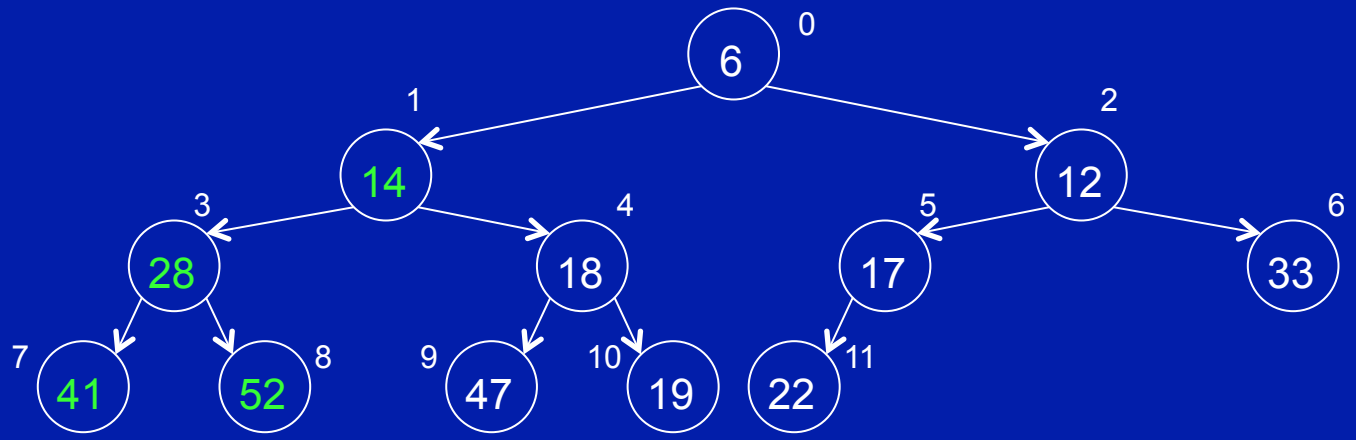
we can find the left child of k at index: $2k + 1$

we can find the right child of k at index: $2k + 2$

we can find the parent of k at index: $\lfloor (k - 1) / 2 \rfloor$

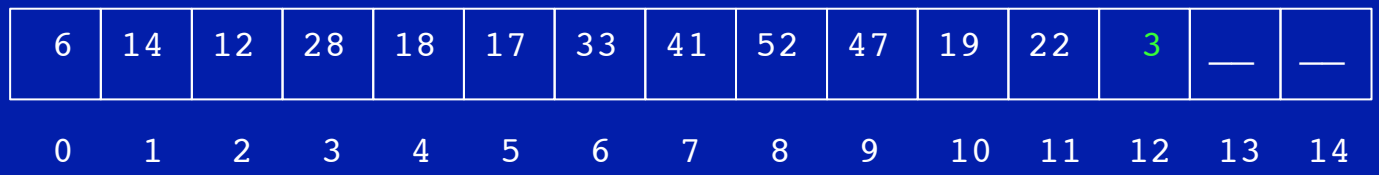
If the node we're looking at is 28, then k = 3
The left child is 41 at index = 7
The right child is 52 at index = 8
The parent is 14 at index = 1

Implementing a heap



If the node we're looking at is 28, then k = 3
The left child is 41 at index = 7
The right child is 52 at index = 8
The parent is 14 at index = 1

Implementing a heap



Now we can navigate the heap with simple index arithmetic instead of link traversals. Given a node at index k,

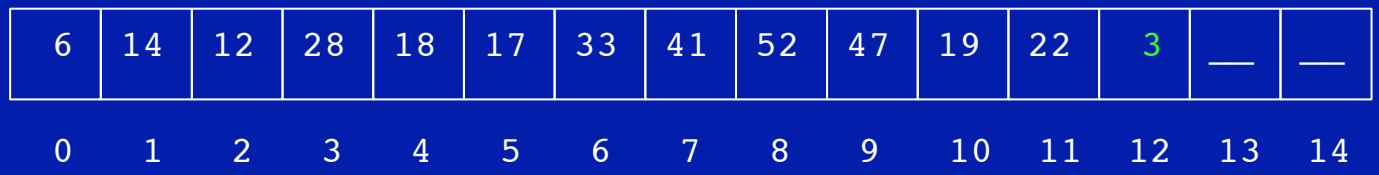
we can find the left child of k at index: $2k + 1$

we can find the right child of k at index: $2k + 2$

we can find the parent of k at index: $\lfloor (k - 1) / 2 \rfloor$

Let's insert 3

Implementing a heap



Now we can navigate the heap with simple index arithmetic instead of link traversals. Given a node at index k,

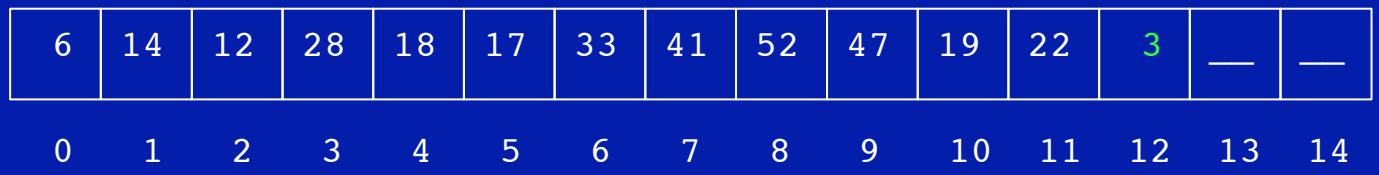
we can find the left child of k at index: $2k + 1$

we can find the right child of k at index: $2k + 2$

we can find the parent of k at index: $\lfloor (k - 1) / 2 \rfloor$

Let's insert 3
Less than or equal to its parent? (index = 5, value = 17)

Implementing a heap



Now we can navigate the heap with simple index arithmetic instead of link traversals. Given a node at index k,

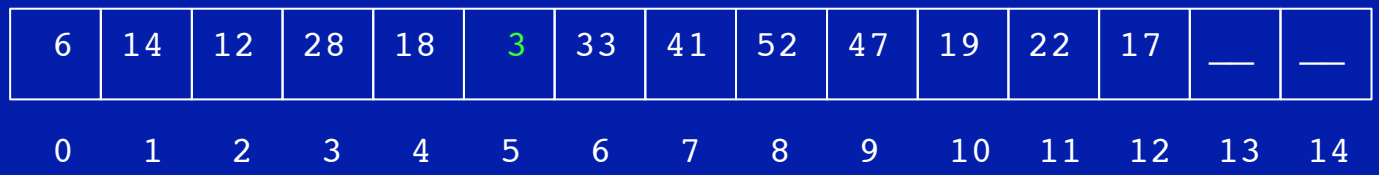
we can find the left child of k at index: $2k + 1$

we can find the right child of k at index: $2k + 2$

we can find the parent of k at index: $\lfloor (k - 1) / 2 \rfloor$

Let's insert 3
Less than or equal to its parent? (index = 5, value = 17) Yes, swap them

Implementing a heap



Now we can navigate the heap with simple index arithmetic instead of link traversals. Given a node at index k,

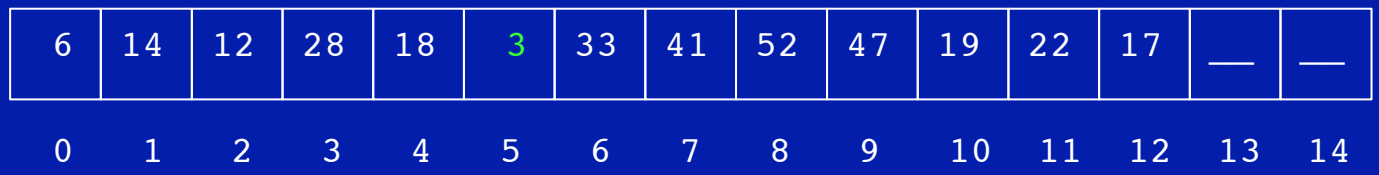
we can find the left child of k at index: $2k + 1$

we can find the right child of k at index: $2k + 2$

we can find the parent of k at index: $\lfloor (k - 1) / 2 \rfloor$

Let's insert 3
Less than or equal to its parent? (index = 5, value = 17) Yes, swap them

Implementing a heap



Now we can navigate the heap with simple index arithmetic instead of link traversals. Given a node at index k,

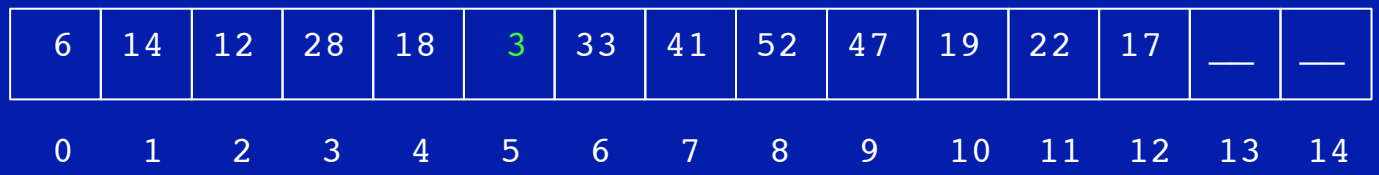
we can find the left child of k at index: $2k + 1$

we can find the right child of k at index: $2k + 2$

we can find the parent of k at index: $\lfloor (k - 1) / 2 \rfloor$

Let's insert 3
Less than or equal to its parent? (index = 5, value = 17) Yes, swap them
Less than or equal to its parent? (index = 2, value = 12)

Implementing a heap



Now we can navigate the heap with simple index arithmetic instead of link traversals. Given a node at index k,

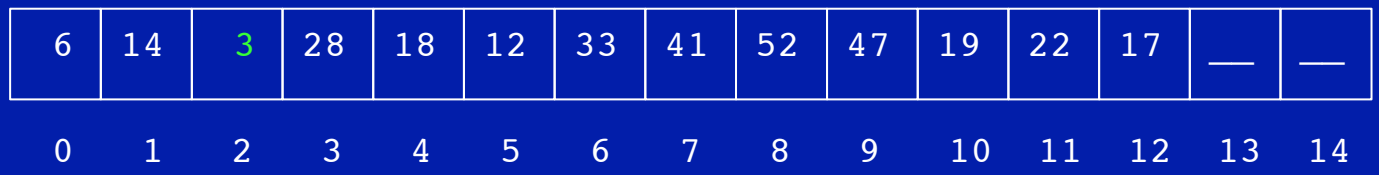
we can find the left child of k at index: $2k + 1$

we can find the right child of k at index: $2k + 2$

we can find the parent of k at index: $\lfloor (k - 1) / 2 \rfloor$

Let's insert 3
Less than or equal to its parent? (index = 5, value = 17) Yes, swap them
Less than or equal to its parent? (index = 2, value = 12) Yes, swap them

Implementing a heap



Now we can navigate the heap with simple index arithmetic instead of link traversals. Given a node at index k,

we can find the left child of k at index: $2k + 1$

we can find the right child of k at index: $2k + 2$

we can find the parent of k at index: $\lfloor (k - 1) / 2 \rfloor$

Let's insert 3
Less than or equal to its parent? (index = 5, value = 17) Yes, swap them
Less than or equal to its parent? (index = 2, value = 12) Yes, swap them

Implementing a heap

6	14	3	28	18	12	33	41	52	47	19	22	17	__	__
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Now we can navigate the heap with simple index arithmetic instead of link traversals. Given a node at index k,

we can find the left child of k at index: $2k + 1$

we can find the right child of k at index: $2k + 2$

we can find the parent of k at index: $\lfloor (k - 1) / 2 \rfloor$

Let's insert 3
Less than or equal to its parent? (index = 5, value = 17) Yes, swap them
Less than or equal to its parent? (index = 2, value = 12) Yes, swap them
Less than or equal to its parent? (index = 0, value = 6)

Implementing a heap

6	14	3	28	18	12	33	41	52	47	19	22	17	__	__
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Now we can navigate the heap with simple index arithmetic instead of link traversals. Given a node at index k,

we can find the left child of k at index: $2k + 1$

we can find the right child of k at index: $2k + 2$

we can find the parent of k at index: $\lfloor (k - 1) / 2 \rfloor$

Let's insert 3
Less than or equal to its parent? (index = 5, value = 17) Yes, swap them
Less than or equal to its parent? (index = 2, value = 12) Yes, swap them
Less than or equal to its parent? (index = 0, value = 6) Yes, swap them

Implementing a heap

3	14	6	28	18	12	33	41	52	47	19	22	17	__	__
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Now we can navigate the heap with simple index arithmetic instead of link traversals. Given a node at index k,

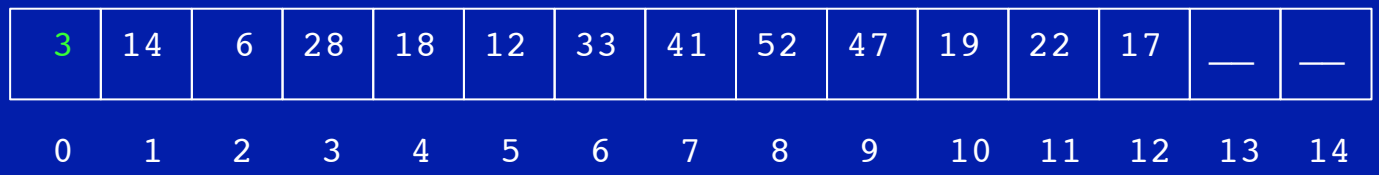
we can find the left child of k at index: $2k + 1$

we can find the right child of k at index: $2k + 2$

we can find the parent of k at index: $\lfloor (k - 1) / 2 \rfloor$

Let's insert 3
Less than or equal to its parent? (index = 5, value = 17) Yes, swap them
Less than or equal to its parent? (index = 2, value = 12) Yes, swap them
Less than or equal to its parent? (index = 0, value = 6) Yes, swap them

Implementing a heap



Now we can navigate the heap with simple index arithmetic instead of link traversals. Given a node at index k,

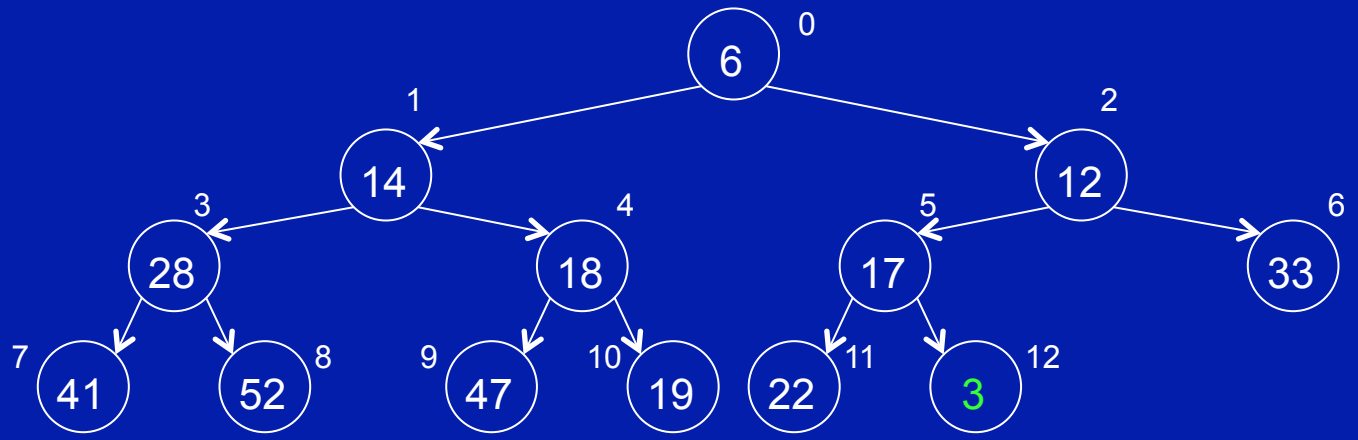
we can find the left child of k at index: $2k + 1$

we can find the right child of k at index: $2k + 2$

we can find the parent of k at index: $\lfloor (k - 1) / 2 \rfloor$

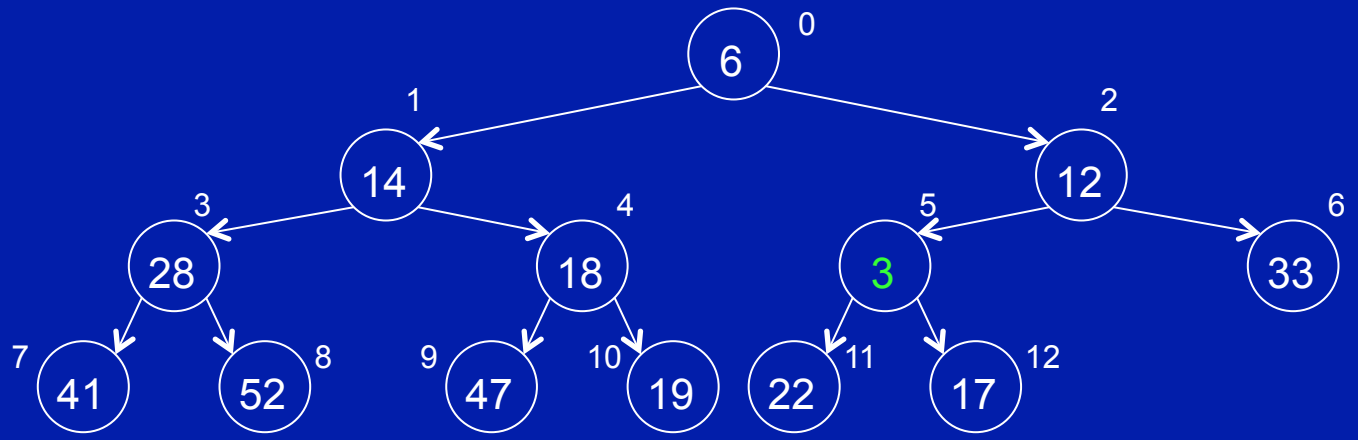
Let's insert 3
Less than or equal to its parent? (index = 5, value = 17) Yes, swap them
Less than or equal to its parent? (index = 2, value = 12) Yes, swap them
Less than or equal to its parent? (index = 0, value = 6) Yes, swap them
Less than or equal to its parent? Wait, we're at top of the heap – done!

Implementing a heap



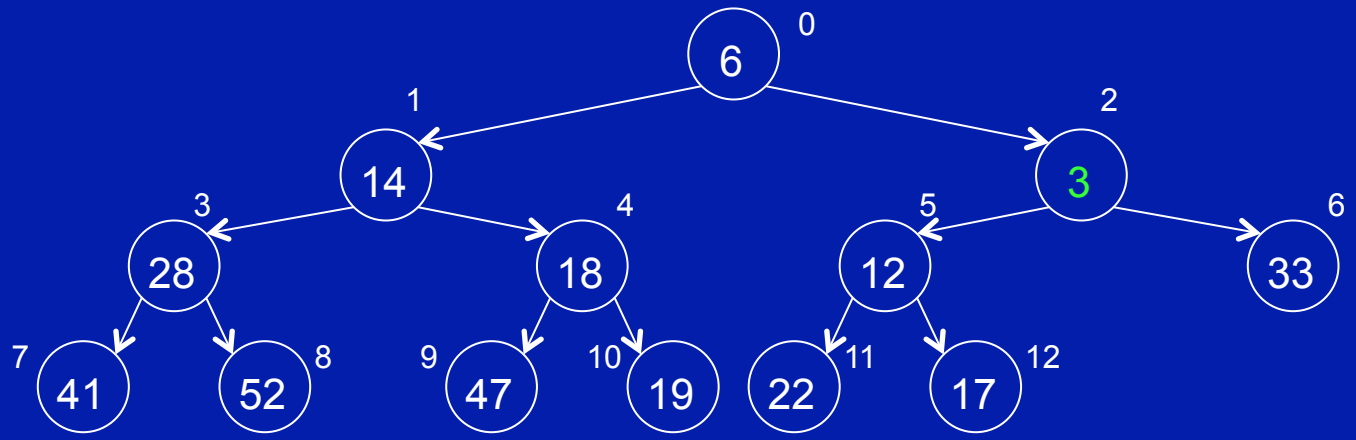
Let's insert 3
Less than or equal to its parent? (index = 5, value = 17) Yes, swap them
Less than or equal to its parent? (index = 2, value = 12) Yes, swap them
Less than or equal to its parent? (index = 0, value = 6) Yes, swap them
Less than or equal to its parent? Wait, we're at top of the heap – done!

Implementing a heap



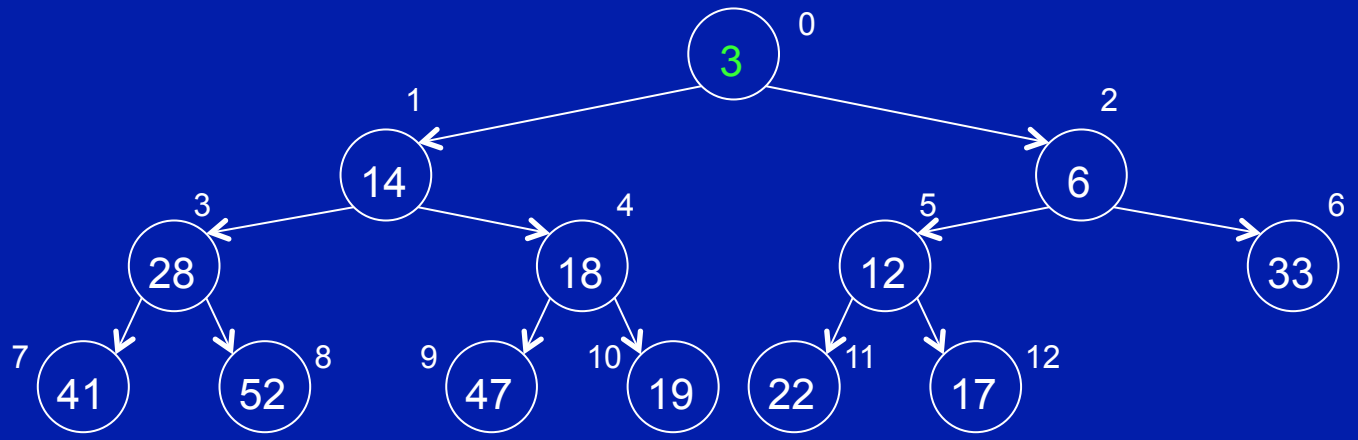
Let's insert 3
Less than or equal to its parent? (index = 5, value = 17) Yes, swap them
Less than or equal to its parent? (index = 2, value = 12) Yes, swap them
Less than or equal to its parent? (index = 0, value = 6) Yes, swap them
Less than or equal to its parent? Wait, we're at top of the heap – done!

Implementing a heap



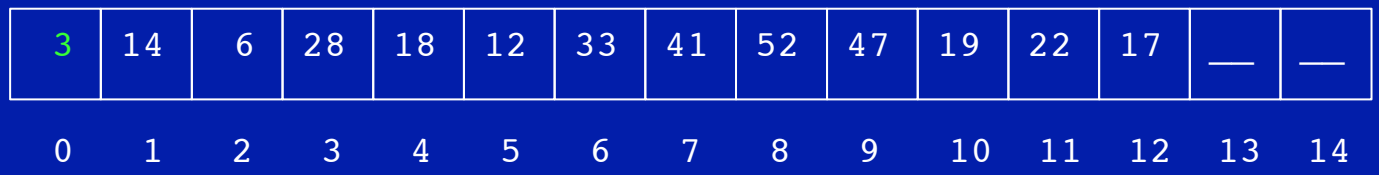
Let's insert 3
Less than or equal to its parent? (index = 5, value = 17) Yes, swap them
Less than or equal to its parent? (index = 2, value = 12) Yes, swap them
Less than or equal to its parent? (index = 0, value = 6) Yes, swap them
Less than or equal to its parent? Wait, we're at top of the heap – done!

Implementing a heap



Let's insert 3
Less than or equal to its parent? (index = 5, value = 17) Yes, swap them
Less than or equal to its parent? (index = 2, value = 12) Yes, swap them
Less than or equal to its parent? (index = 0, value = 6) Yes, swap them
Less than or equal to its parent? Wait, we're at top of the heap – done!

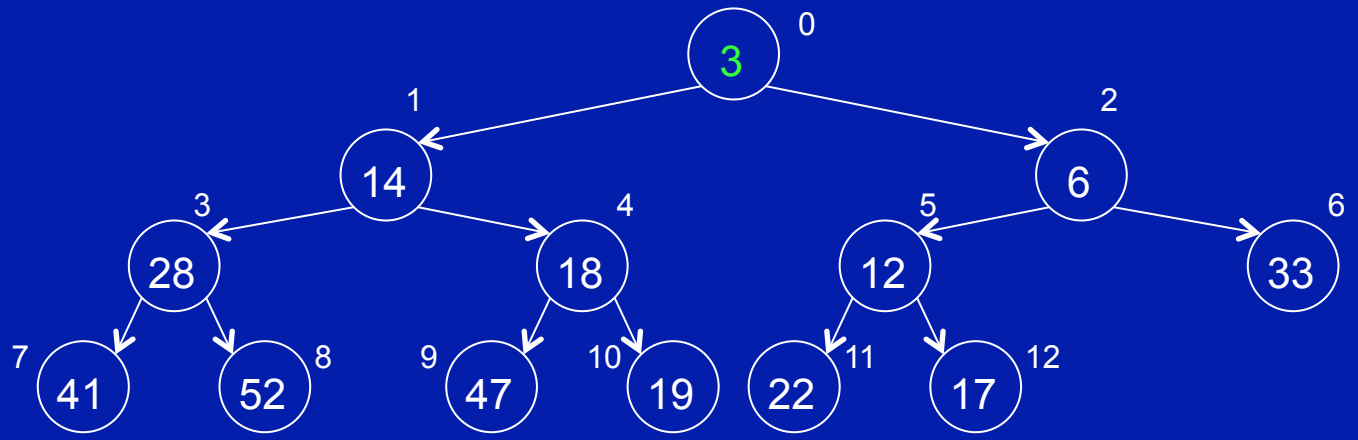
Implementing a heap



The more formal algorithms for insertion in and deletion from a heap are in your textbook. You should make sure you understand the details.

Priority queues

The heap data structure provides the basis for the priority queue abstract data type. If we assume that a value in the heap represents the priority of a job (smaller numbers having higher priority, like number of pages in a print request), then as values are added, the smaller, higher priority values float toward the top, and lower priority values fall to the bottom.

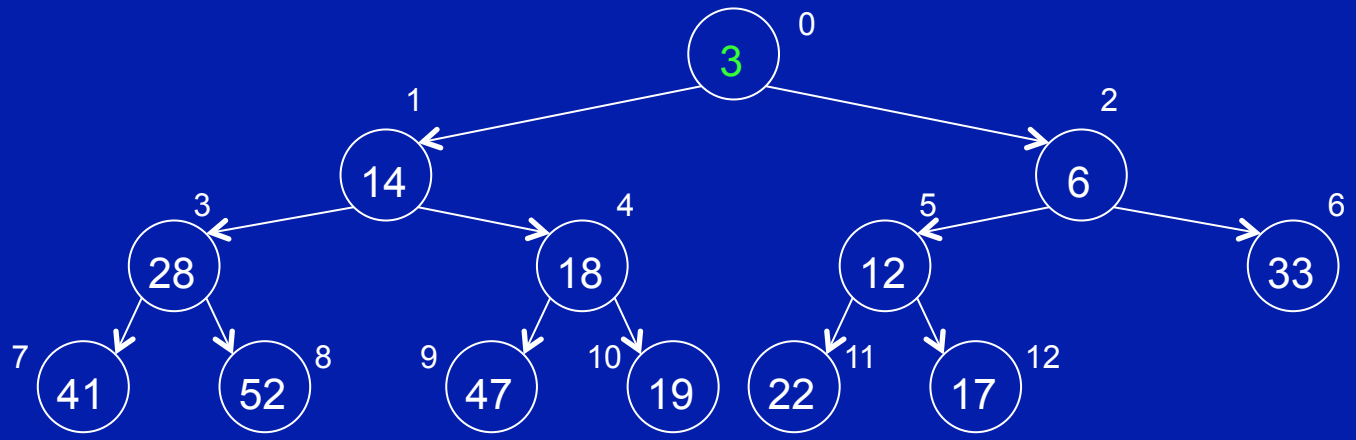


When we want to select the highest priority job in the queue, we just grab the item at the top of the heap (e.g. 3). We then move the last item in the heap to the top and perform ReheapDown as described previously.

Priority queues

Removing the minimum value (i.e., highest priority item) from the priority queue requires ReheapDown, which takes O(lg n) time.

Adding a new item to the priority queue requires ReheapUp, which also takes O(lg n) time.



Heapsort

We can use a heap as the foundation for a very efficient sorting algorithm called heapsort.

Heapsort consists of two phases:

Heapify: build a heap using the elements to be sorted

Sort: Use the heap to sort the data

This can all be done in place in the array that holds the heap, but it's easier to see if we draw the trees instead of the array. Once you see how it works with trees, make sure you understand how it works in place in the array.

Heapsort

Here's the heapify component:

for each item in the sequence to be sorted
    add the item to the next available position in the complete binary tree
    restore the heap property (using ReheapUp)

Say we want to sort the sequence 5 2 1 4 3:

Heapsort

Here's the heapify component:

for each item in the sequence to be sorted
    add the item to the next available position in the complete binary tree
    restore the heap property (using ReheapUp)

Say we want to sort the sequence 5 2 1 4 3:

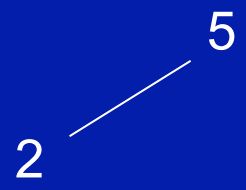
5

Heapsort

Here's the heapify component:

for each item in the sequence to be sorted
    add the item to the next available position in the complete binary tree
    restore the heap property (using ReheapUp)

Say we want to sort the sequence 5 2 1 4 3:

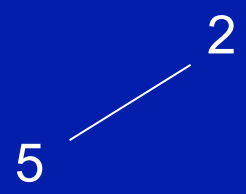


Heapsort

Here's the heapify component:

for each item in the sequence to be sorted
    add the item to the next available position in the complete binary tree
    restore the heap property (using ReheapUp)

Say we want to sort the sequence 5 2 1 4 3:

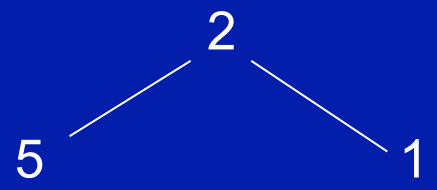


Heapsort

Here's the heapify component:

for each item in the sequence to be sorted
    add the item to the next available position in the complete binary tree
    restore the heap property (using ReheapUp)

Say we want to sort the sequence 5 2 1 4 3:

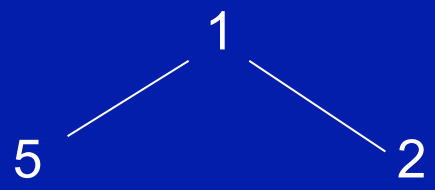


Heapsort

Here's the heapify component:

for each item in the sequence to be sorted
    add the item to the next available position in the complete binary tree
    restore the heap property (using ReheapUp)

Say we want to sort the sequence 5 2 1 4 3:

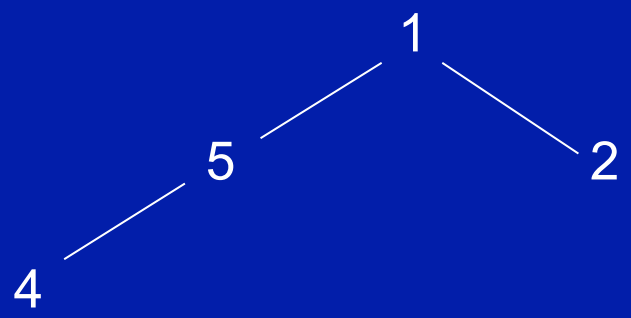


Heapsort

Here's the heapify component:

for each item in the sequence to be sorted
    add the item to the next available position in the complete binary tree
    restore the heap property (using ReheapUp)

Say we want to sort the sequence 5 2 1 4 3:

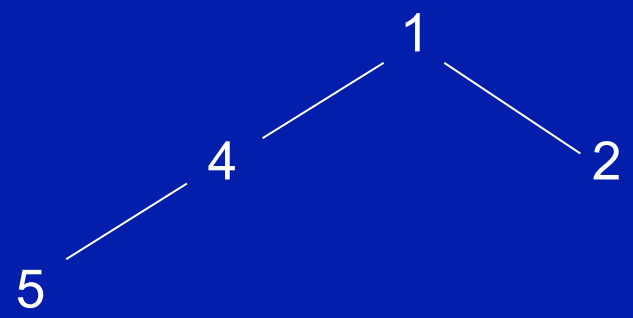


Heapsort

Here's the heapify component:

for each item in the sequence to be sorted
    add the item to the next available position in the complete binary tree
    restore the heap property (using ReheapUp)

Say we want to sort the sequence 5 2 1 4 3:

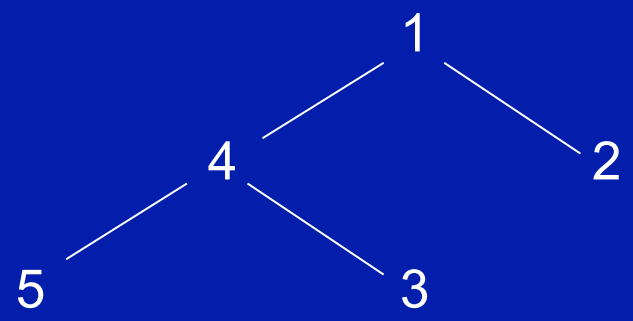


Heapsort

Here's the heapify component:

for each item in the sequence to be sorted
    add the item to the next available position in the complete binary tree
    restore the heap property (using ReheapUp)

Say we want to sort the sequence 5 2 1 4 3:

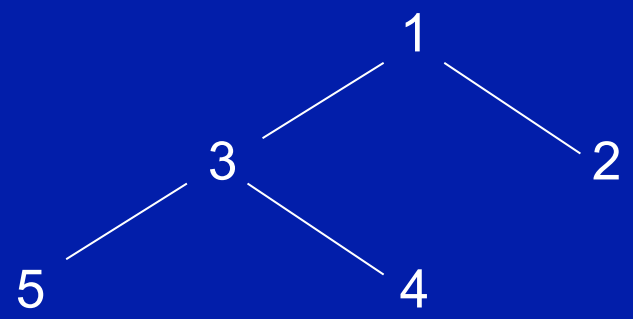


Heapsort

Here's the heapify component:

for each item in the sequence to be sorted
    add the item to the next available position in the complete binary tree
    restore the heap property (using ReheapUp)

Say we want to sort the sequence 5 2 1 4 3:

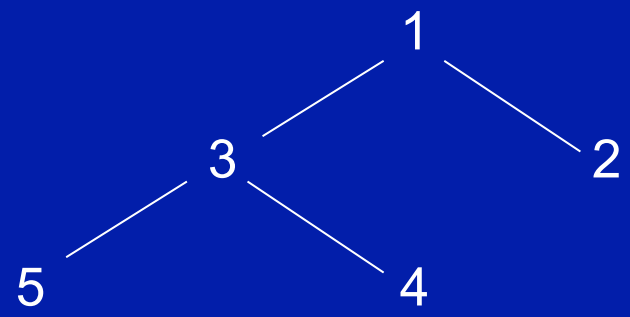


Heapsort

Here's the heapify component:

for each item in the sequence to be sorted
    add the item to the next available position in the
        complete binary tree
    restore the heap property (using ReheapUp)

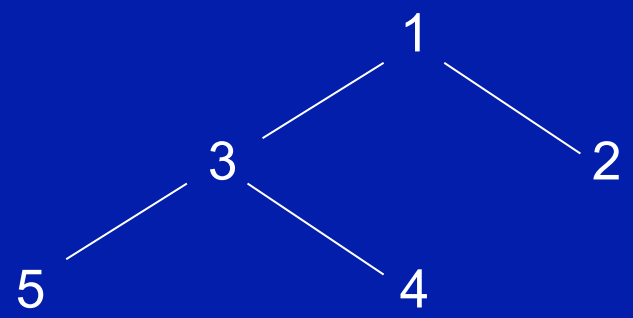
Say we want to sort the sequence 5 2 1 4 3: The sequence is heapified



Heapsort

Now for the sorting:

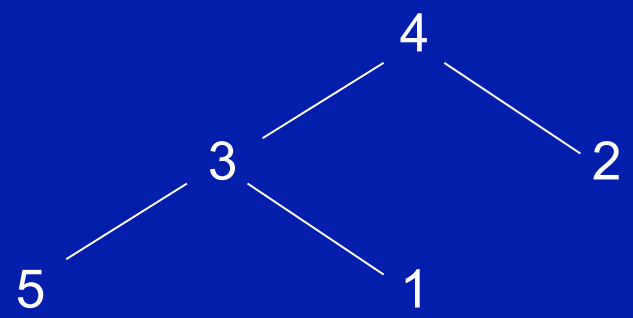
while the heap is not empty
    remove the first item from the heap by swapping it
        with the last item in the heap
    reduce the size of the heap by one
    restore the heap property



Heapsort

Now for the sorting:

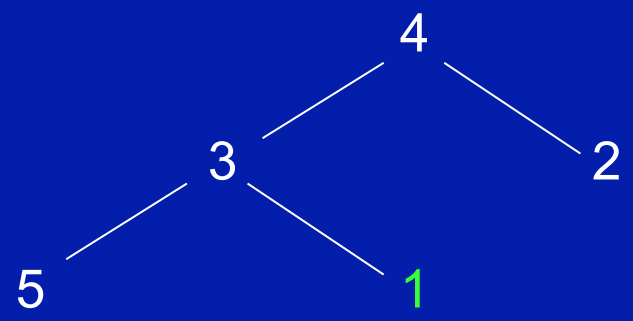
while the heap is not empty
    remove the first item from the heap by swapping it
        with the last item in the heap
    reduce the size of the heap by one
    restore the heap property



Heapsort

Now for the sorting:

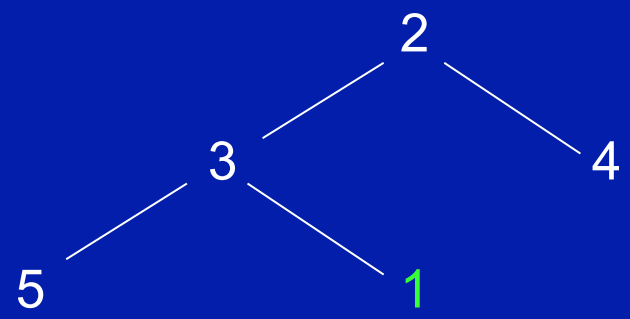
while the heap is not empty
    remove the first item from the heap by swapping it
        with the last item in the heap
    reduce the size of the heap by one
    restore the heap property



Heapsort

Now for the sorting:

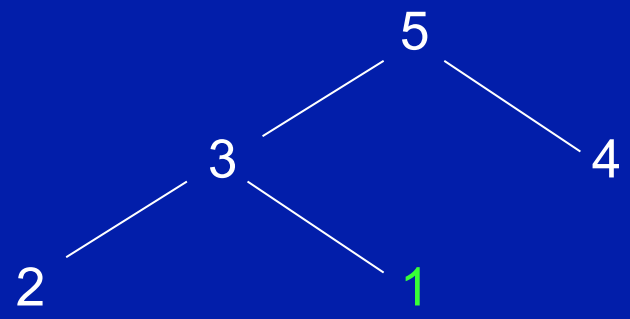
while the heap is not empty
    remove the first item from the heap by swapping it
        with the last item in the heap
    reduce the size of the heap by one
    restore the heap property



Heapsort

Now for the sorting:

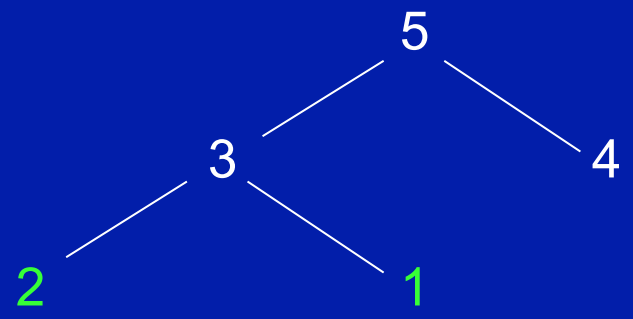
while the heap is not empty
    remove the first item from the heap by swapping it
        with the last item in the heap
    reduce the size of the heap by one
    restore the heap property



Heapsort

Now for the sorting:

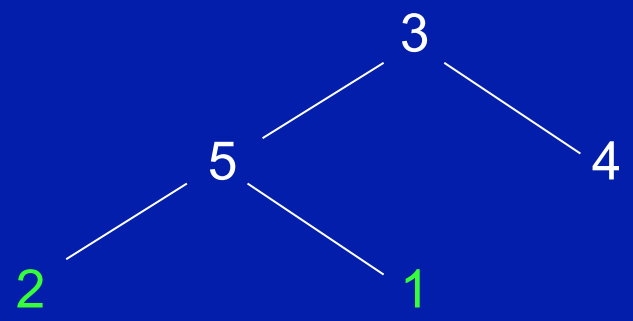
while the heap is not empty
    remove the first item from the heap by swapping it
        with the last item in the heap
    reduce the size of the heap by one
    restore the heap property



Heapsort

Now for the sorting:

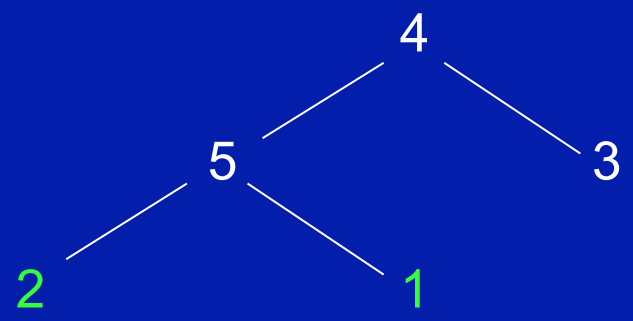
while the heap is not empty
    remove the first item from the heap by swapping it
        with the last item in the heap
    reduce the size of the heap by one
    restore the heap property



Heapsort

Now for the sorting:

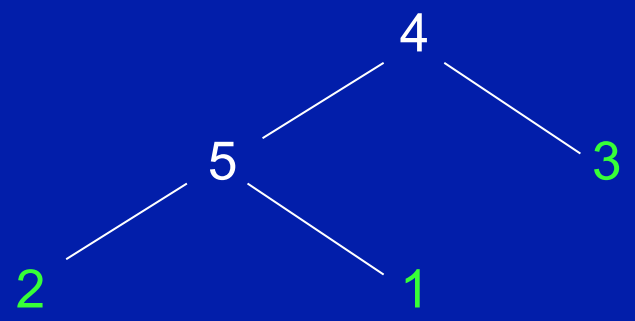
while the heap is not empty
    remove the first item from the heap by swapping it
        with the last item in the heap
    reduce the size of the heap by one
    restore the heap property



Heapsort

Now for the sorting:

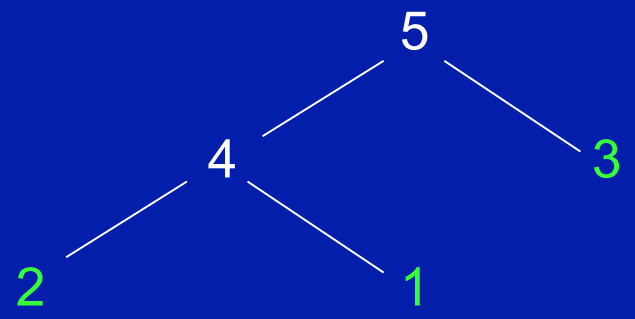
while the heap is not empty
    remove the first item from the heap by swapping it
        with the last item in the heap
    reduce the size of the heap by one
    restore the heap property



Heapsort

Now for the sorting:

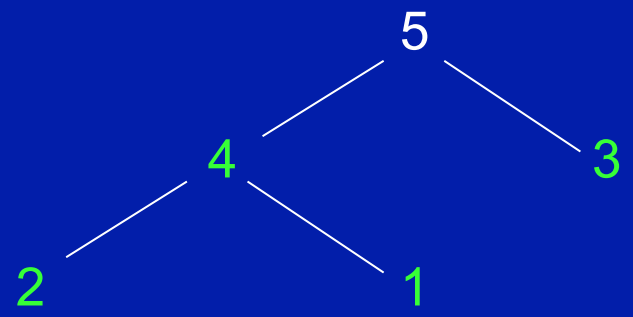
while the heap is not empty
    remove the first item from the heap by swapping it
        with the last item in the heap
    reduce the size of the heap by one
    restore the heap property



Heapsort

Now for the sorting:

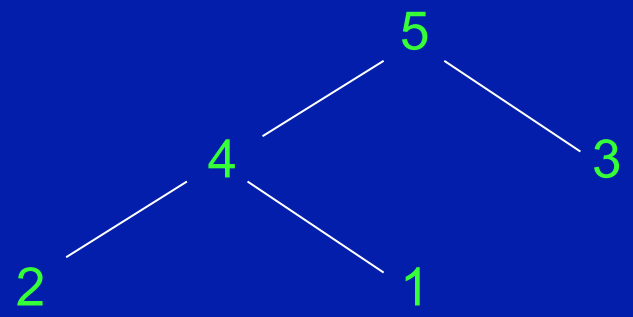
while the heap is not empty
    remove the first item from the heap by swapping it
        with the last item in the heap
    reduce the size of the heap by one
    restore the heap property



Heapsort

Now for the sorting:

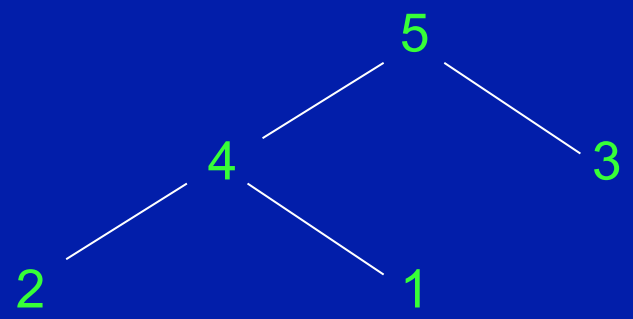
while the heap is not empty
    remove the first item from the heap by swapping it
        with the last item in the heap
    reduce the size of the heap by one
    restore the heap property



Heapsort

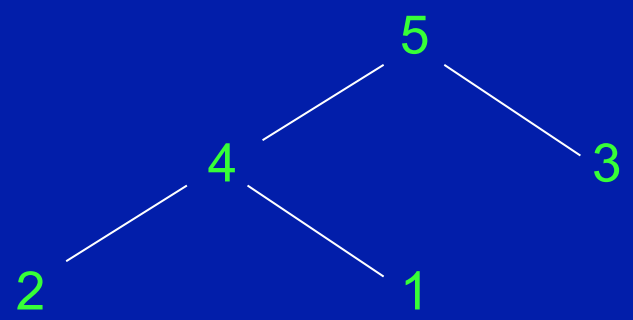
Now it's sorted, but it's largest to smallest (reading top to bottom, left to right).

We created a min heap, and we ended up with the sequence sorted from largest to smallest. The same thing will happen when sorting in place in the array.



Heapsort

In the array-based heapsort example in the book (which you should study), the original sequence was heapified into a max heap, and the resulting sorted sequence in the array went from smallest to largest. Just something to keep in mind.



Heapsort analysis

A heap of size n has lg n levels. Building a heap requires finding the correct location for each item in a heap with lg n levels. Because there are n items to insert in the heap and each insert is O(lg n), the time to heapify the original unsorted sequence using ReheapUp or Sift-up is O(n lg n). During sorting, we have n items to remove from the heap, which then is also O(n lg n). Because we can do it all in the orginal array, no extra storage is required.