one that is to be deleted. Replace the left-most node by its right subtree, if this exists; otherwise just delete it.

The last part works because the left-most node in the right sub-tree is guaranteed to be bigger than all nodes in the left sub-tree, smaller than all the other nodes in the right sub-tree, and have no left sub-tree itself. For instance, if we delete the node with value `11` from the tree in Figure 6.1, we get the tree displayed in Figure 7.1.

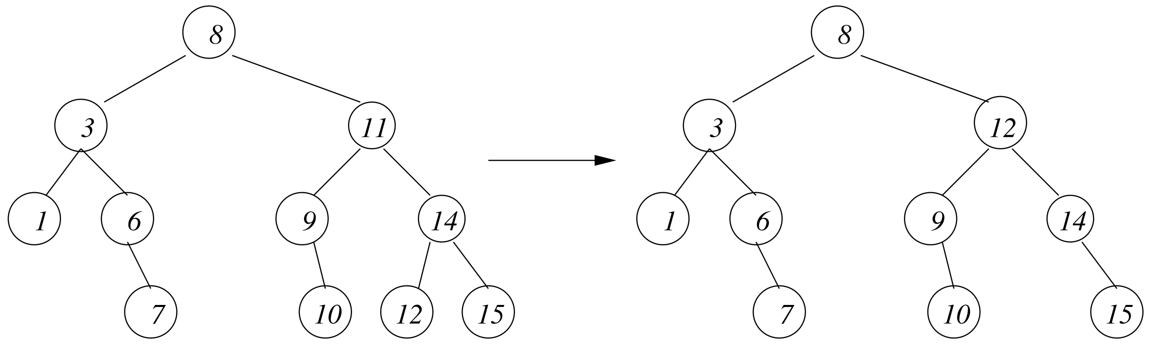


Figure 7.1: Example of node deletion in a binary search tree.

In practice, we need to turn the above algorithm (specified in words) into a more detailed algorithm specified using the primitive binary tree operators:

```
delete(value v, tree t) {
  if ( isEmpty(t) )
     error('Error: given item is not in given tree')
  else
     if ( v < root(t) )    // delete from left sub-tree
        return MakeTree(root(t), delete(v,left(t)), right(t));
     else if ( v > root(t) )     // delete from right sub-tree
        return MakeTree(root(t), left(t), delete(v,right(t)));
     else    // the item v to be deleted is root(t)
        if ( isEmpty(left(t)) )
           return right(t)
        elseif ( isEmpty(right(t)) )
           return left(t)
        else    // difficult case with both subtrees non-empty
           return MakeTree(smallestNode(right(t)), left(t),
                                          removeSmallestNode(right(t))
}
```

If the empty tree condition is met, it means the search item is not in the tree, and an appropriate error message should be returned.

The `delete` procedure uses two sub-algorithms to find and remove the smallest item of a given sub-tree. Since the relevant sub-trees will always be non-empty, these sub-algorithms can be written with that *precondition*. However, it is always the responsibility of the programmer to ensure that any preconditions are met whenever a given procedure is used, so it is important to say explicitly what the preconditions are. It is often safest to start each procedure with a check to determine whether the preconditions are satisfied, with an appropriate *error message* produced when they are not, but that may have a significant time cost if the procedure is called many times. First, to find the smallest node, we have:

```
smallestNode(tree t) {
  // Precondition: t is a non-empty binary search tree
  if ( isEmpty(left(t) )
     return root(t)
  else
     return smallestNode(left(t));
}
```

which uses the fact that, by the definition of a binary search tree, the smallest node of `t` is the left-most node. It recursively looks in the left sub-tree till it reaches an empty tree, at which point it can return the root. The second sub-algorithm uses the same idea:

```
removeSmallestNode(tree t) {
  // Precondition: t is a non-empty binary search tree
  if ( isEmpty(left(t) )
     return right(t)
  else
     return MakeTree(root(t), removeSmallestNode(left(t)), right(t))
}
```

except that the remaining tree is returned rather than the smallest node.

These procedures are further examples of recursive algorithms. In each case, the recursion is guaranteed to terminate, because every recursive call involves a smaller tree, which means that we will eventually find what we are looking for or reach an empty tree.

It is clear from the algorithm that the deletion of a node requires the same number of steps as searching for a node, or inserting a new node, i.e. the average height of the binary search tree, or $O(\log_2 n)$ where n is the total number of nodes on the tree.

7.8 Checking whether a binary tree is a binary search tree

Building and using binary search trees as discussed above is usually enough. However, another thing we sometimes need to do is *check* whether or not a given binary tree is a binary search tree, so we need an algorithm to do that. We know that an empty tree is a (trivial) binary search tree, and also that all nodes in the left sub-tree must be smaller than the root and themselves form a binary search tree, and all nodes in the right sub-tree must be greater than the root and themselves form a binary search tree. Thus the obvious algorithm is:

```
isbst(tree t) {
   if ( isEmpty(t) )
      return true
   else
      return ( allsmaller(left(t),root(t)) and isbst(left(t))
                 and allbigger(right(t),root(t)) and isbst(right(t)) )
}
```

```
allsmaller(tree t, value v) {
   if ( isEmpty(t) )
      return true
   else
      return ( (root(t) < v) and allsmaller(left(t),v)
                                       and allsmaller(right(t),v) )
}

allbigger(tree t, value v) {
   if ( isEmpty(t) )
      return true
   else
      return ( (root(t) > v) and allbigger(left(t),v)
                                       and allbigger(right(t),v) )
}
```

However, the simplest or most obvious algorithm is not always the most efficient. Exercise: identify what is inefficient about this algorithm, and formulate a more efficient algorithm.

7.9 Sorting using binary search trees

Sorting is the process of putting a collection of items in order. We shall formulate and discuss many sorting algorithms later, but we are already able to present one of them.

The node values stored in a binary search tree can be printed in ascending order by recursively printing each left sub-tree, root, and right sub-tree in the right order as follows:

```
printInOrder(tree t) {
   if ( not isEmpty(t) ) {
      printInOrder(left(t))
      print(root(t))
      printInOrder(right(t))
   }
}
```

Then, if the collection of items to be sorted is given as an array `a` of known size `n`, they can be printed in sorted order by the algorithm:

```
sort(array a of size n) {
   t = EmptyTree
   for i = 0,1,...,n-1
      t = insert(a[i],t)
   printInOrder(t)
}
```

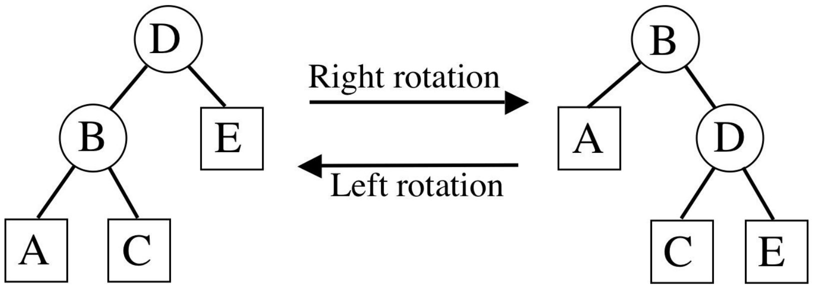
which starts with an empty tree, inserts all the items into it using `insert(v,t)` to give a binary search tree, and then prints them in order using `printInOrder(t)`. Exercise: modify this algorithm so that instead of printing the sorted values, they are put back into the original array in ascending order.

7.10 Balancing binary search trees

If the items are added to a binary search tree in random order, the tree tends to be fairly well balanced with height not much more than $\log_2 n$. However, there are many situations where the added items are not in random order, such as when adding new student IDs. In the extreme case of the new items being added in ascending order, the tree will be one long branch off to the right, with height $n \gg \log_2 n$.

If all the items to be inserted into a binary search tree are already sorted, it is straightforward to build a perfectly balanced binary tree from them. One simply has to recursively build a binary tree with the middle (i.e., median) item as the root, the left subtree made up of the smaller items, and the right subtree made up of the larger items. This idea can be used to *rebalance* any existing binary search tree, because the existing tree can easily be output into a sorted array as discussed in Section 7.9. Exercise: Write an algorithm that rebalances a binary search tree in this way, and work out its time complexity.

Another way to avoid unbalanced binary search trees is to *rebalance* them from time to time using *tree rotations*. Such tree rotations are best understood as follows: Any binary search tree containing at least two nodes can clearly be drawn in one of the two forms:



where B and D are the required two nodes to be rotated, and A, C and E are binary search sub-trees (any of which may be empty). The two forms are related by left and right tree rotations which clearly preserve the binary search tree property. In this case, any nodes in sub-tree A would be shifted up the tree by a right rotation, and any nodes in sub-tree E would be shifted up the tree by a left rotation. For example, if the left form had A consisting of two nodes, and C and E consisting of one node, the height of the tree would be reduced by one and become *perfectly balanced* by a right tree rotation.

Typically, such tree rotations would need to be applied to many different sub-trees of a full tree to make it perfectly balanced. For example, if the left form had C consisting of two nodes, and A and E consisting of one node, the tree would be balanced by first performing a left rotation of the A-B-C sub-tree, followed by a right rotation of the whole tree. In practice, finding suitable sequences of appropriate tree rotations to rebalance an arbitrary binary search tree is not straightforward, but it is possible to formulate systematic balancing algorithms that are more efficient than outputting the whole tree and rebuilding it.

7.11 Self-balancing AVL trees

Self-balancing binary search trees avoid the problem of unbalanced trees by automatically *rebalancing* the tree throughout the insertion process to keep the height close to $\log_2 n$ at each stage. Obviously, there will be a cost involved in such rebalancing, and there will be a trade-off between the time involved in rebalancing and the time saved by the reduced height of the tree, but generally it is worthwhile.

The earliest type of self-balancing binary search tree was the *AVL tree* (named after its inventors G.M. Adelson-Velskii and E.M. Landis). These maintain the difference in heights of the two sub-trees of all nodes to be at most one. This requires the tree to be periodically rebalanced by performing one or more *tree rotations* as discussed above, but the complexity of insertion, deletion and search remain at $O(\log_2 n)$.

The general idea is to keep track of the *balance factor* for each node, which is the height of the left sub-tree minus the height of the right sub-tree. By definition, all the nodes in an AVL-tree will have a balance factor in the integer range $[-1, 1]$. However, insertion or deletion of a node could leave that in the wider range $[-2, 2]$ requiring a tree-rotation to bring it back into AVL form. Exercise: Find some suitable algorithms for performing efficient AVL tree rotations. Compare them with other self-balancing approaches such as *red-black trees*.

7.12 B-trees

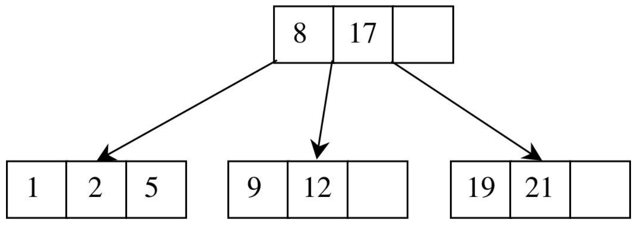
A *B-tree* is a generalization of a self-balancing binary search tree in which each node can hold more than one search key and have more than two children. The structure is designed to allow more efficient self-balancing, and offers particular advantages when the node data needs to be kept in external storage such as disk drives. The standard (Knuth) definition is:

Definition. A *B-tree* of order m is a tree which satisfies the following conditions:

- Every node has at most m children.
- Every non-leaf node (except the root node) has at least $m/2$ children.
- The root node, if it is not a leaf node, has at least two children.
- A non-leaf node with c children contains $c-1$ search keys which act as separation values to divide its sub-trees.
- All leaf nodes appear in the same level, and carry information.

There appears to be no definitive answer to the question of what the "B" in "B-Tree" stands for. It is certainly not "Binary", but it could equally well be "balanced", "broad" or "bushy", or even "Boeing" because they were invented by people at Boeing Research Labs.

The standard representation of simple order 4 example with 9 search keys would be:



The search keys held in each node are ordered (e.g., 1, 2, 5 in the example), and the non-leaf node's search keys (i.e., the items 8 and 17 in the example) act as separation values to divide the contents of its sub-trees in much the same way that a node's value in a binary search tree separates the values held in its two sub-trees. For example, if a node has 3 child nodes (or sub-trees) then it must have 2 separation values $s1$ and $s2$. All values in the leftmost subtree will be less than $s1$, all values in the middle subtree will be between $s1$ and $s2$, and all values in the rightmost subtree will be greater than $s2$. That allows insertion and searching to proceed from the root down in a similar way to binary search trees.

The restriction on the number of children to lie between $m/2$ and m means that the best case height of an order m B-tree containing n search keys is $\log_m n$ and the worst case height is $\log_{m/2} n$. Clearly the costs of insertion, deletion and searching will all be proportional to the tree height, as in a binary search tree, which makes them very efficient. The requirement that all the leaf nodes are at the same level means that B-trees are always balanced and thus have minimal height, though *rebalancing* will often be required to restore that property after insertions and deletions.

The *order* of a B-tree is typically chosen to optimize a particular application and implementation. To maintain the conditions of the B-tree definition, non-leaf nodes often have to be split or joined when new items are inserted into or deleted from the tree (which is why there is a factor of two between the minimum and maximum number of children), and *rebalancing* is often required. This renders the insertion and deletion algorithms somewhat more complicated than for binary search trees. An advantage of B-trees over self balancing binary search trees, however, is that the range of child nodes means that rebalancing is required less frequently. A disadvantage is that there may be more space wastage because nodes will rarely be completely full. There is also the cost of keeping the items within each node ordered, and having to search among them, but for reasonably small orders m, that cost is low. Exercise: find some suitable insertion, deletion and rebalancing algorithms for B-trees.

Chapter 8

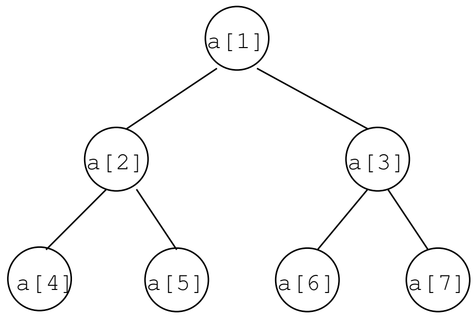
Priority Queues and Heap Trees

8.1 Trees stored in arrays

It was noted earlier that binary trees can be *stored* with the help of *pointer*-like structures, in which each item contains references to its children. If the tree in question is a *complete* binary tree, there is a useful *array* based alternative.

Definition. A binary tree is *complete* if every level, except possibly the last, is completely filled, and all the leaves on the last level are placed as far to the left as possible.

Intuitively, a complete binary tree is one that can be obtained by filling the nodes starting with the root, and then each next level in turn, always from the left, until one runs out of nodes. Complete binary trees always have minimal height for their size n, namely $\log_2 n$, and are always *perfectly balanced* (but not every perfectly balanced tree is complete in the sense of the above definition). Moreover, and more importantly, it is possible for them to be stored straightforwardly in arrays, top-to-bottom left-to-right, as in the following example:



For complete binary trees, such arrays provide very tight representations.

Notice that this time we have chosen to start the array with index 1 rather than 0. This has several computational advantages. The nodes on level i then have indices $2^i, \cdots, 2^{i+1}-1$. The level of a node with index i is $\lfloor \log_2 i \rfloor$, that is, $\log_2 i$ rounded down. The children of a node with index i, if they exist, have indices $2i$ and $2i+1$. The parent of a child with index i has index $i/2$ (using integer division). This allows the following simple algorithms:

```
boolean isRoot(int i) {
   return i == 1
}
```

```
int level(int i) {
   return log(i)
}

int parent(int i) {
   return i / 2
}

int left(int i) {
   return 2 * i
}

int right(int i) {
   return 2 * i + 1
}
```

which make the processing of these trees much easier.

This way of storing a binary tree as an array, however, will not be efficient if the tree is not complete, because it involves reserving space in the array for every possible node in the tree. Since keeping binary search trees balanced is a difficult problem, it is therefore not really a viable option to adapt the algorithms for binary search trees to work with them stored as arrays. Array-based representations will also be inefficient for binary search trees because node insertion or deletion will usually involve shifting large portions of the array. However, we shall now see that there is another kind of binary tree for which array-based representations allow very efficient processing.

8.2 Priority queues and binary heap trees

While most queues in every-day life operate on a first come, first served basis, it is sometimes important to be able to assign a *priority* to the items in the queue, and always serve the item with the highest priority next. An example of this would be in a hospital casualty department, where life-threatening injuries need to be treated first. The structure of a complete binary tree in array form is particularly useful for representing such *priority queues*.

It turns out that these queues can be implemented efficiently by a particular type of complete binary tree known as a *binary heap tree*. The idea is that the node labels, which were the search keys when talking about binary search trees, are now numbers representing the priority of each item in question (with higher numbers meaning a higher priority in our examples). With heap trees, it is possible to insert and delete elements efficiently without having to keep the whole tree sorted like a binary search tree. This is because we only ever want to remove one element at a time, namely the one with the highest priority present, and the idea is that the highest priority item will always be found at the root of the tree.

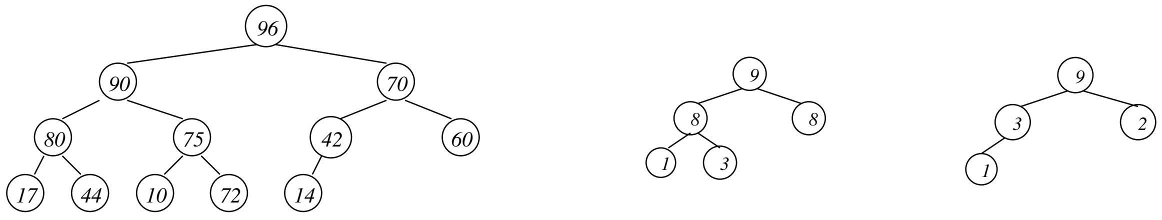
Definition. A *binary heap tree* is a complete binary tree which is either empty or satisfies the following conditions:

- The priority of the root is higher than (or equal to) that of its children.
- The left and right subtrees of the root are heap trees.

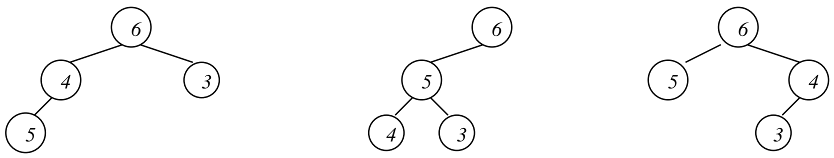
Alternatively, one could define a heap tree as a complete binary tree such that the priority of every node is higher than (or equal to) that of all its descendants. Or, as a complete binary tree for which the priorities become smaller along every path down through the tree.

The most obvious difference between a binary heap tree and a binary search trees is that the biggest number now occurs at the root rather than at the right-most node. Secondly, whereas with binary search trees, the left and right sub-trees connected to a given parent node play very different rôles, they are interchangeable in binary heap trees.

Three examples of binary trees that are valid heap trees are:



and three which are not valid heap trees are:



the first because $5 > 4$ violates the required priority ordering, the second because it is not perfectly balanced and hence not complete, and the third because it is not complete due to the node on the last level not being as far to the left as possible.

8.3 Basic operations on binary heap trees

In order to develop algorithms using an array representation, we need to allocate memory and keep track of the largest position that has been filled so far, which is the same as the current number of nodes in the heap tree. This will involve something like:

```
int MAX = 100     // Maximum number of nodes allowed
int heap[MAX+1]   // Stores priority values of nodes of heap tree
int n = 0         // Largest position that has been filled so far
```

For heap trees to be a useful representation of priority queues, we must be able to *insert* new nodes (or customers) with a given priority, *delete* unwanted nodes, and identify and remove the top-priority node, i.e. the *root* (that is, 'serve' the highest priority customer). We also need to be able to determine when the queue/tree is empty. Thus, assuming the priorities are given by integers, we need a constructor, mutators/selectors, and a condition:

```
insert(int p, array heap, int n)
delete(int i, array heap, int n)
int root(array heap, int n)
boolean heapEmpty(array heap, int n)
```

Identifying whether the heap tree is empty, and getting the root and last leaf, is easy:

```
boolean heapEmpty(array heap, int n) {
   return n == 0
}

int root(array heap, int n) {
   if ( heapEmpty(heap,n) )
      error('Heap is empty')
   else return heap[1]
}

int lastLeaf(array heap, int n) {
   if ( heapEmpty(heap,n) )
      error('Heap is empty')
   else return heap[n]
}
```

Inserting and deleting heap tree nodes is also straightforward, but not quite so easy.

8.4 Inserting a new heap tree node

Since we always keep track of the last position `n` in the tree which has been filled so far, we can easily insert a new element at position $\mathtt{n} + \mathtt{1}$, provided there is still room in the array, and increment `n`. The tree that results will still be a complete binary tree, but the heap tree priority ordering property might have been violated. Hence we may need to '*bubble up*' the new element into a valid position. This can be done easily by comparing its priority with that of its parent, and if the new element has higher priority, then it is exchanged with its parent. We may have to repeat this process, but once we reach a parent that has higher or equal priority, we can stop because we know there can be no lower priority items further up the tree. Hence an algorithm which inserts a new heap tree node with priority `p` is:

```
insert(int p, array heap, int n) {
   if ( n == MAX ) {
      error('Heap is full')
   else {
      heap[n+1] = p
      bubbleUp(n+1,heap,n+1)
   }
 }

bubbleUp(int i, array heap, int n) {
   if ( isRoot(i) )
      return
   elseif ( heap[i] > heap[parent(i)] ) {
      swap heap[i] and heap[parent(i)]
      bubbleUp(parent(i),heap,n)
   }
}
```