Efficient storing and retrieving: trees

Alexandra Silva and Henk Barendregt

{alexandra,henk}@cs.ru.nl
http://www.cs.ru.nl/~{alexandra,henk}

Institute for Computing and Information Sciences
Radboud University Nijmegen

28 February 2012

Recap

Last week's message

- the running time of some algorithms, notably divide and conquer algorithms, can be expressed in the form of a recurrence
- in order to get the time bound complexity the recurrence needs to be solved, using, for instance, the substitution method
- more generally, how to analyze the execution time of a recursive algorithm.

Today

- sorting in linear time
- binary search trees
- red-black trees

How fast can we sort?

We will prove a lower bound, then beat it by playing a different game.

Comparison sorting

- The only operation that may be used to gain order information about a sequence is comparison of pairs of elements.
- All sorts seen so far are comparison sorts: insertion sort, merge sort, quicksort, (selection sort, heapsort, treesort).

Lower bounds for sorting

Lower bounds

- $\Omega(n)$ to examine all the input.
- Best sorts seen so far are $\Omega(n \lg n)$.

We will show that $\Omega(n \lg n)$ is a lower bound for comparison sorts.

Lower bounds for sorting

Lower bounds

- $\Omega(n)$ to examine all the input.
- Best sorts seen so far are $\Omega(n \lg n)$.

We will show that $\Omega(n \lg n)$ is a lower bound for comparison sorts.

Decision tree

Abstraction of (the execution of) any comparison sort. Represents comparisons made by

- a specific sorting algorithm
- on inputs of a given size.

Abstracts away everything else: control and data movement. We are counting only comparisons.

Decision tree

In a decision tree, each node:

- corresponds to a comparison test between two elements in the sequence
- the left (resp. right) sub-tree corresponds to the continuation of executing the algorithm if the test is true (resp. false).

Each leaf in the tree corresponds to a possible ordering of the input sequence. All the permutations of the sequence appear as leaves (why?)

Decision tree

In a decision tree, each node:

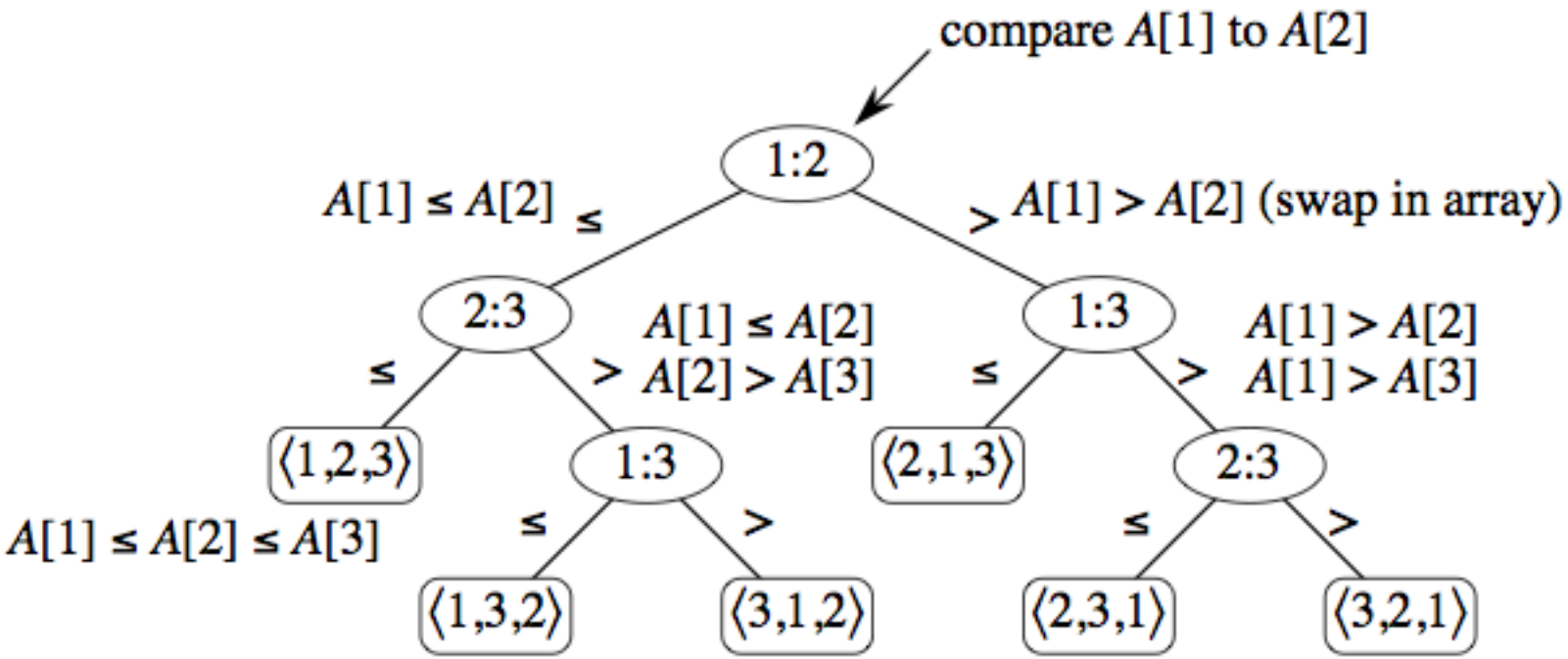
- corresponds to a comparison test between two elements in the sequence
- the left (resp. right) sub-tree corresponds to the continuation of executing the algorithm if the test is true (resp. false).

Each leaf in the tree corresponds to a possible ordering of the input sequence. All the permutations of the sequence appear as leaves (why?)
A concrete execution of the algorithm for a given input vector corresponds to a path from the root of the tree to a leaf (that is, a sequence of comparisons).

The worst case corresponds to the longest path in the tree from the root to a leaf. The number of comparisons is the height of the tree.

Example

For insertion sort on 3 elements:



Worst case

How many leaves on the decision tree?

Worst case

How many leaves on the decision tree?

There are $\geq n!$ leaves, because every permutation appears at least once.

For any comparison sort,

- 1 tree for each n.
- View the tree as if the algorithm splits in two at each node, based on the information it has determined up to that point. The tree models all possible execution traces.
- What is the length of the longest path from root to leaf?

Depends on the algorithm...
Insertion sort: $\Theta(n^2)$, Merge sort: $\Theta(n \lg n)$.

What is the optimal for comparison algorithms?

Lemma

Any binary tree of height h has $\leq 2^h$ leaves. In other words:

- *$l = \#$ of leaves,*
- *$h =$height,*

Then $l \leq 2^h$.

(We'll prove this lemma later.) Why is this useful?

What is the optimal for comparison algorithms?

Lemma

Any binary tree of height h has $\leq 2^h$ leaves. In other words:

- *$l = \#$ of leaves,*
- *$h =$height,*

Then $l \leq 2^h$.

(We'll prove this lemma later.) Why is this useful?

Theorem

Any decision tree that sorts n elements has height $\Omega(n \lg n)$.

What is the optimal for comparison algorithms?

Proof of the theorem

- $l \geq n!$
- Using the lemma above, $n! \leq l \leq 2^h$, that is $2^h \geq n!$
- Take logs: $h \geq \lg(n!)$
- Use Stirling's approximation: $n! > (n/e)^n$

$$\begin{aligned} h &\geq \lg((n/e)^n) \\ &= n \lg(n/e) \\ &= n(\lg n - \lg e) \\ &= \Omega(n \lg n). \end{aligned}$$

What is the optimal for comparison algorithms?

Corollary

Merge sort is an asymptotically optimal comparison sorting algorithm.

Another example you might know: heapsort.

Proof of the aux. lemma

Any binary tree of height h has $\leq 2^h$ *leaves.*

Proof. By induction on h.

Basis: $h = 0$. Tree is just one node, which is a leaf. $2^h = 1$.

Proof of the aux. lemma

Any binary tree of height h has $\leq 2^h$ *leaves.*

Proof. By induction on h.

Basis: $h = 0$. Tree is just one node, which is a leaf. $2^h = 1$.

Inductive step: Assume true for height $= h - 1$. Extend tree of height $h - 1$ by making as many new leaves as possible. Each leaf becomes parent to two new leaves.

$$\#\text{ of leaves for height } h \quad = \quad 2 \times (\#\text{ of leaves for height } h - 1)$$

Proof of the aux. lemma

Any binary tree of height h has $\leq 2^h$ leaves.

Proof. By induction on h.

Basis: $h = 0$. Tree is just one node, which is a leaf. $2^h = 1$.

Inductive step: Assume true for height $= h - 1$. Extend tree of height $h - 1$ by making as many new leaves as possible. Each leaf becomes parent to two new leaves.

$$\begin{aligned} \#\text{ of leaves for height } h &= 2 \times (\#\text{ of leaves for height } h-1) \\ &= 2 \times 2^{h-1} \qquad \text{(ind. hypothesis)} \\ &= 2^h. \end{aligned}$$

Sorting in linear time

Non-comparison sorts.

Counting sort

- Depends on a key assumption: numbers to be sorted are integers in a known interval $0, 1, \ldots, k$.
- The algorithm is based on counting, for each element x in the sequence, of the number of elements smaller or equal to x.
- This counting process allows to determine for each element its final position (why?)

Sorting in linear time

Non-comparison sorts.

Counting sort

- Depends on a key assumption: numbers to be sorted are integers in a known interval $0, 1, \ldots, k$.
- The algorithm is based on counting, for each element x in the sequence, of the number of elements smaller or equal to x.
- This counting process allows to determine for each element its final position (why?)
- The algorithm returns a sorted output vector and uses an auxiliary storage

Sorting in linear time

Non-comparison sorts.

Counting sort

- Depends on a key assumption: numbers to be sorted are integers in a known interval $0, 1, \ldots, k$.
- The algorithm is based on counting, for each element x in the sequence, of the number of elements smaller or equal to x.
- This counting process allows to determine for each element its final position (why?)
- The algorithm returns a sorted output vector and uses an auxiliary storage

The algorithm does not resort to comparisons and hence can be better than $\Omega(n \lg n)$!

Counting sort

```
void counting_sort(int A[], int B[], int k) {
int C[k+1];
for (i=0 ; i<=k ; i++)  /* initializing C[] */
  C[i] = 0;
for (j=1 ; j<=N ; j++) /* counting A[j] */
  C[A[j]] = C[A[j]]+1;

for (i=1 ; i<=k ; i++) /* counting <= i */
  C[i] = C[i]+C[i-1];

for (j=N ; j>=1 ; j--) { /* build sorted array */
  B[C[A[j]]] = A[j];
  C[A[j]] = C[A[j]]-1;
 }
}
```

What is the role of the second line in the last cycle?

Counting sort – analysis

```
void counting_sort(int A[], int B[], int k) {
int C[k+1];
for (i=0 ; i<=k ; i++)
  C[i] = 0;
for (j=1 ; j<=N ; j++)
  C[A[j]] = C[A[j]]+1;

for (i=1 ; i<=k ; i++)
  C[i] = C[i]+C[i-1];

for (j=N ; j>=1 ; j--) {
  B[C[A[j]]] = A[j];
  C[A[j]] = C[A[j]]-1;
 }
}
```

$\Theta(k)$

$\Theta(N)$

$\Theta(k)$

$\Theta(N)$

$\Theta(N+k)$

Counting sort – analysis

```
void counting_sort(int A[], int B[], int k) {
int C[k+1];
for (i=0 ; i<=k ; i++)                          Θ(k)
  C[i] = 0;
for (j=1 ; j<=N ; j++)                          Θ(N)
  C[A[j]] = C[A[j]]+1;

for (i=1 ; i<=k ; i++)                          Θ(k)
  C[i] = C[i]+C[i-1];

for (j=N ; j>=1 ; j--) {                        Θ(N)
  B[C[A[j]]] = A[j];
  C[A[j]] = C[A[j]]-1;
 }                                              Θ(N + k)
}
```

If $k = O(N)$ then $T(N) = \Theta(N + k) = \Theta(N)$. Linear time!

Counting sort – side remarks

How big a k is practical?

- Good for sorting 32-bit values? No.
- 16-bit? Probably not.
- 8-bit? Maybe, depending on n.
- 4-bit? Probably (unless n is really small).

Counting sort will be used in radix sort.

Counting sort – side remarks

How big a k is practical?

- Good for sorting 32-bit values? No.
- 16-bit? Probably not.
- 8-bit? Maybe, depending on n.
- 4-bit? Probably (unless n is really small).

Counting sort will be used in radix sort.

Counting sort is stable (keys with same value appear in same order in output as they did in input; why?)

Counting sort – side remarks

How big a k is practical?

- Good for sorting 32-bit values? No.
- 16-bit? Probably not.
- 8-bit? Maybe, depending on n.
- 4-bit? Probably (unless n is really small).

Counting sort will be used in radix sort.

Counting sort is stable (keys with same value appear in same order in output as they did in input; why?) because of how the last loop works.

(stability is only important when there is date associated with the keys)

Radix sort

How IBM made its money. Punch card readers for census tabulation in early 1900's. Card sorters, worked on one column at a time. It's the algorithm for using the machine that extends the technique to multi-column sorting. The human operator was part of the algorithm!

Radix sort

Algorithm is useful for sequences with multiple keys. Imagine an array with dates (day, month, year).
To sort it we can

- Write a comparison function that first compares the years, then, if equal, the months, then if equal, the days

$$\begin{aligned} 10/2/1984 &< 10/2/2012 \quad \text{only years} \\ 10/2/1984 &< 10/3/1984 \quad \text{years and months} \\ 9/2/1984 &< 10/2/1984 \quad \text{all} \end{aligned}$$

  With the comparison function, then any sorting algorithm can be used.

Radix sort

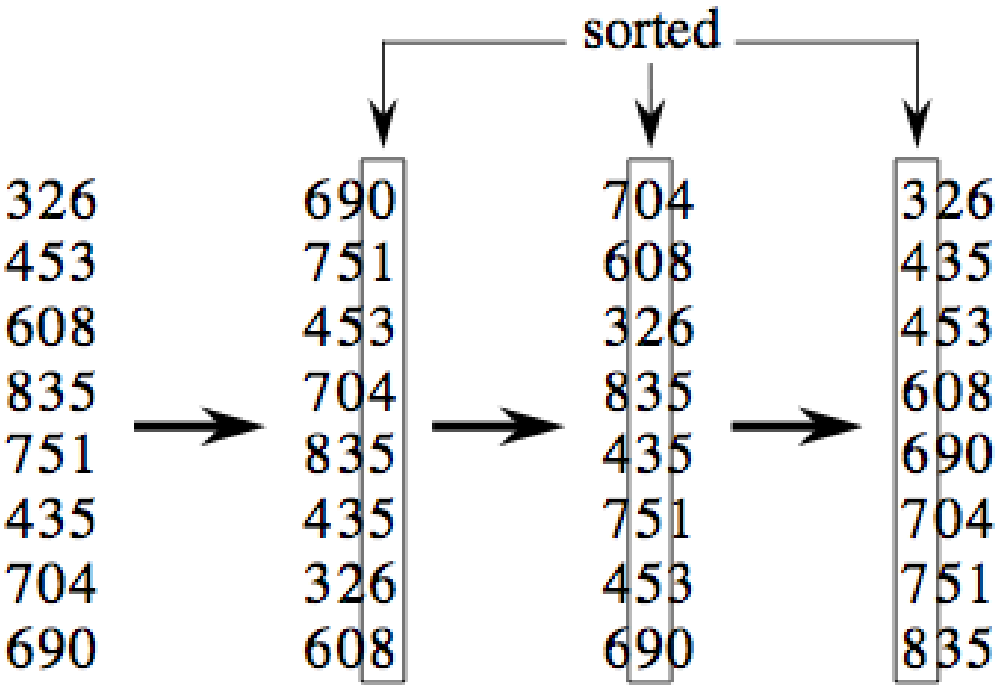
Algorithm is useful for sequences with multiple keys. Imagine an array with dates (day, month, year).
To sort it we can

- Write a comparison function that first compares the years, then, if equal, the months, then if equal, the days

  $10/2/1984 < 10/2/2012$ only years
  $10/2/1984 < 10/3/1984$ years and months
  $9/2/1984 < 10/2/1984$ all

  With the comparison function, then any sorting algorithm can be used.
- We could instead sort the sequence three times, each time for each key. For that, we need the sorting algorithm to be **stable**.

Example


sorted
326
453
608
835
751
435
704
690
690
751
453
704
835
435
326
608
704
608
326
835
435
751
453
690
326
435
453
608
690
704
751
835

Radix Sort

```
void radix_sort(int A[], int p, int r, int d) {
 for (i=1; i<=d; i++)
   stable_sort_by_index(i, A, p, r);
}
```

- the least significant digit is 1, the most significant is d.
- the algorithm `stable_sort_by_index(i, A, p, r)`:
  - sorts the vector $A[p..r]$, using as key the digit i;
  - has to be stable (e.g, counting sort, and also merge sort, insertion sort, ...).

Radix sort – analysis

Assume that we use counting sort as the intermediate sort.

- $\Theta(n + k)$ per pass (digits in range $0, \ldots, k$)
- d passes
- $\Theta(d(n + k))$ total

If $k = O(n)$, time $= \Theta(dn)$.
If d is small enough then time $= \Theta(n)$.

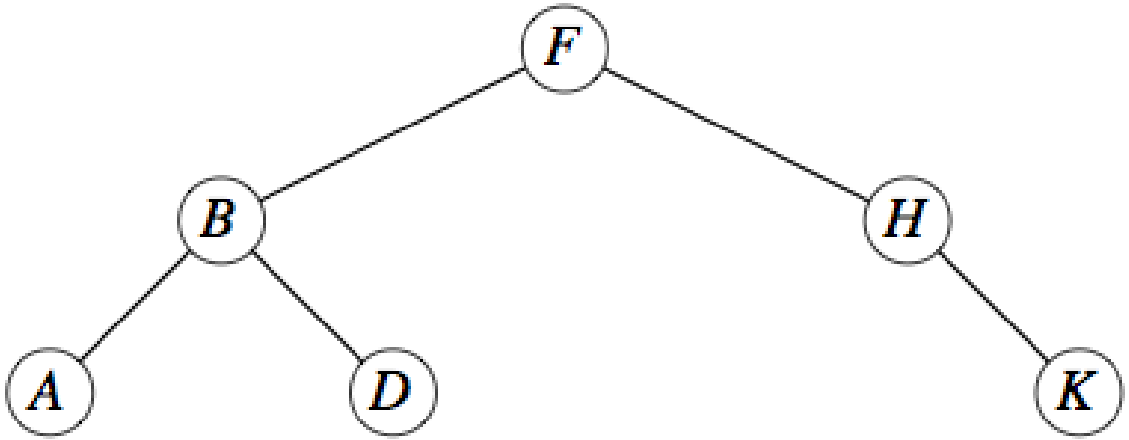
Search trees

- Data structures that support many dynamic-set operations.
- Can be used as both a dictionary and as a priority queue.
- Basic operations take time proportional to the height of the tree.
  - For complete binary tree with n nodes: worst case $\Theta(\lg n)$.
  - For linear chain of n nodes: worst case $\Theta(n)$.
- Different types of search trees include binary search trees, red-black trees

Binary search trees

- Binary search trees are an important data structure for dynamic sets.
- Accomplish many dynamic-set operations in O(h) time, where h = height of tree.
- Each node of the tree contains the fields
  - *key* (and possibly other satellite data).
  - *left*: points to left child.
  - *right*: points to right child.
  - *p*: points to parent.
- Stored keys must satisfy the binary-search-tree property.
  - If y is in left subtree of x, then $key[y] \leq key[x]$.
  - If y is in right subtree of x, then $key[y] \geq key[x]$.

Example



Binary search property:

$$\begin{array}{rcccl} B, A, D & \leq & F & \leq & H, K \\ A & \leq & B & \leq & D \\ & & H & \leq & K \end{array}$$

Some operations on binary search trees

- The *binary-search-tree property* allows us to print keys in a binary search tree in order
- Recursively, using an algorithm called an inorder tree walk.
- Elements are printed in monotonically increasing order.

Some operations on binary search trees

- The *binary-search-tree property* allows us to print keys in a binary search tree in order
- Recursively, using an algorithm called an inorder tree walk.
- Elements are printed in monotonically increasing order.

How INORDER-TREE-WALK works:

- Check to make sure that node x has a key.
- Recursively, print the keys of the nodes in x's left subtree. Print x's key.
- Recursively, print the keys of the nodes in x's right subtree.

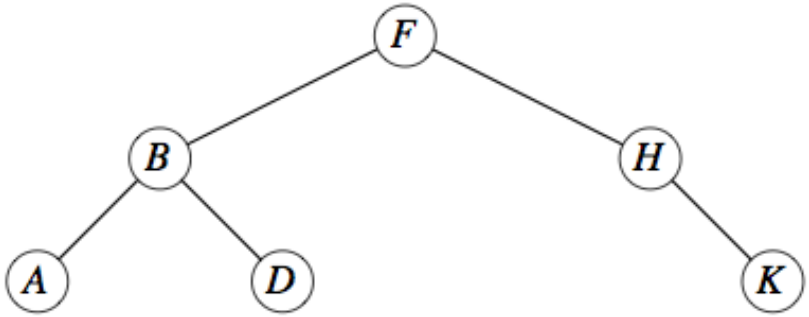
Some operations on binary search trees

```
INORDER-TREE-WALK(x)
if x ≠ NIL
  then INORDER-TREE-WALK(left[x])
       print key[x]
       INORDER-TREE-WALK(right[x])
```

Some operations on binary search trees

```
INORDER-TREE-WALK(x)
if x ≠ NIL
  then INORDER-TREE-WALK(left[x])
       print key[x]
       INORDER-TREE-WALK(right[x])
```

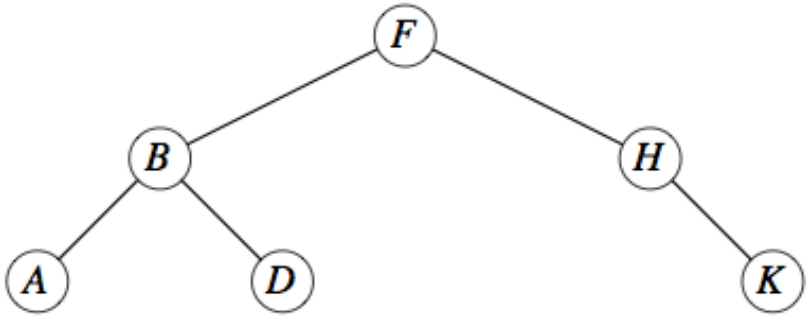
Example



Some operations on binary search trees

```
INORDER-TREE-WALK(x)
if x ≠ NIL
  then INORDER-TREE-WALK(left[x])
       print key[x]
       INORDER-TREE-WALK(right[x])
```

Example

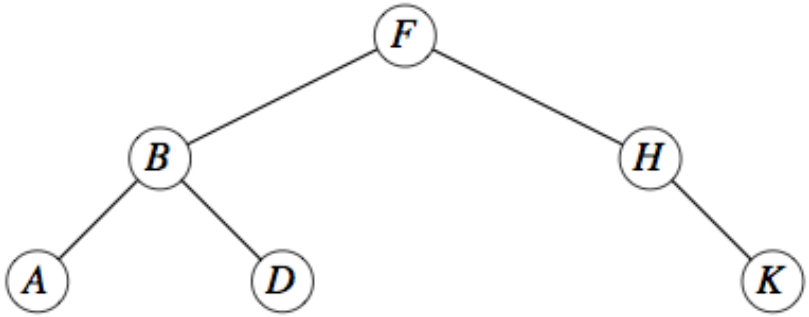


ABDFHK

Some operations on binary search trees

```
INORDER-TREE-WALK(x)
if x ≠ NIL
  then INORDER-TREE-WALK(left[x])
       print key[x]
       INORDER-TREE-WALK(right[x])
```

Example



ABDFHK

The walk takes $\Theta(n)$ time for a tree with n nodes, because we visit and print each node once.

Querying a binary search tree

Because of the *binary search property* we have a dictionary-like search.

```
TREE-SEARCH(x, k)
if x = NIL or k = key[x]
   then return x
if k < key[x]
   then return TREE-SEARCH(left[x], k)
   else  return TREE-SEARCH(right[x], k)
```

Querying a binary search tree

Because of the *binary search property* we have a dictionary-like search.

```
TREE-SEARCH(x, k)
if x = NIL or k = key[x]
   then return x
if k < key[x]
   then return TREE-SEARCH(left[x], k)
   else  return TREE-SEARCH(right[x], k)
```

The algorithm recurses, visiting nodes on a downward path from the root. Thus, running time is $O(h)$, where h is the height of the tree.

Minimum and maximum

The binary-search-tree property guarantees that

- the minimum key of a binary search tree is located at the leftmost node, and
- the maximum key of a binary search tree is located at the rightmost node. Traverse the appropriate sub-trees (left or right) until the end is reached.

```
TREE-MINIMUM(x)
while left[x] ≠ NIL
    do x ← left[x]
return x
```

```
TREE-MAXIMUM(x)
while right[x] ≠ NIL
    do x ← right[x]
return x
```

Minimum and maximum

The binary-search-tree property guarantees that

- the minimum key of a binary search tree is located at the leftmost node, and
- the maximum key of a binary search tree is located at the rightmost node. Traverse the appropriate sub-trees (left or right) until the end is reached.

```
TREE-MINIMUM(x)
while left[x] ≠ NIL
    do x ← left[x]
return x
```

```
TREE-MAXIMUM(x)
while right[x] ≠ NIL
    do x ← right[x]
return x
```

Both procedures visit nodes that form a downward path from the root to a leaf. Both procedures run in $O(h)$ time, where h is the height of the tree.

Further operations

- Insertion: traverse the tree downwards by comparing the value of current node to v (the value to insert), and move to the left or right child accordingly, until finding the right place for v.
- Deletion: find the value to delete; reorganize the left/right subtrees of the parent of v.

Further operations

- Insertion: traverse the tree downwards by comparing the value of current node to v (the value to insert), and move to the left or right child accordingly, until finding the right place for v.
- Deletion: find the value to delete; reorganize the left/right subtrees of the parent of v.

On a tree of height h, both procedures take $O(h)$ time.

Minimizing running time

We have been analyzing running time in terms of h (the height of the binary search tree), instead of n (the number of nodes in the tree).

Problem: Worst case for binary search tree is $\Theta(n)$: no better than a list.
Solution: Guarantee small height (balanced tree) $h = O(\lg n)$.

Minimizing running time

We have been analyzing running time in terms of h (the height of the binary search tree), instead of n (the number of nodes in the tree).

Problem: Worst case for binary search tree is $\Theta(n)$: no better than a list.
Solution: Guarantee small height (balanced tree) $h = O(\lg n)$.

By varying the properties of binary search trees, one is able to analyze running time in terms of n.

Method: Restructure the tree if necessary. Nothing special is required for querying, but there may be extra work when changing the structure of the tree (inserting or deleting).

Minimizing running time

We have been analyzing running time in terms of h (the height of the binary search tree), instead of n (the number of nodes in the tree).

Problem: Worst case for binary search tree is $\Theta(n)$: no better than a list.
Solution: Guarantee small height (balanced tree) $h = O(\lg n)$.

By varying the properties of binary search trees, one is able to analyze running time in terms of n.

Method: Restructure the tree if necessary. Nothing special is required for querying, but there may be extra work when changing the structure of the tree (inserting or deleting).

Red-black trees are a special class of binary trees that avoids the worst-case behavior of $O(n)$ like plain binary search trees.

Red-black trees

- A variation of binary search trees.
- **Balanced**: height is $O(\lg n)$, where n is the number of nodes.

Red-black trees

- A variation of binary search trees.
- **Balanced**: height is $O(\lg n)$, where n is the number of nodes.
- Every node is either red or black.
- The root is black.
- Every leaf is black.
- If a node is red, then both its children are black. (Hence no two reds in a row on a simple path from the root to a leaf.)
- For each node, all paths from the node to descendant leaves contain the same number of black nodes.

Red-black trees

- Operations like minimum, maximum or search run in $O(h)$ time. Thus, they take $O(\lg n)$ time on red-black trees.

Red-black trees

- Operations like minimum, maximum or search run in $O(h)$ time. Thus, they take $O(\lg n)$ time on red-black trees.
- Insertion and deletion are not so easy...
- The tree needs to remain balanced
- If we insert, what color to make the new node? If we delete, thus removing a node, we can violate the red-black property (generating two red nodes in a row).

Red-black trees

- Operations like minimum, maximum or search run in $O(h)$ time. Thus, they take $O(\lg n)$ time on red-black trees.
- Insertion and deletion are not so easy...
- The tree needs to remain balanced
- If we insert, what color to make the new node? If we delete, thus removing a node, we can violate the red-black property (generating two red nodes in a row).
- you will learn (are learning?) the algorithms in another course, but to keep in mind: even with all the sanity checks insertion and deletion are performed in $O(\lg n)$.