Design and Analysis of Algorithms
演算法設計與分析
Lecture 3
March 11, 2010
洪國寶
Homework # 2

1. 3.1-1 (p. 52)
2. 3-1 a, b (p. 61) (for O and Ω)
3. 4.3-7 (p. 87)
4. 4.4-3 (p. 93)

Due March 17, 2010
Outline

• Review
• Recurrences (cont.)
• The maximum subarray problem
• Elementary data structures
• Hash tables
• Binary search trees
Review: Merge sort

• Merge Sort uses the *divide-and-conquer paradigm*:
  – *Divide* the problem into smaller subproblems
  – *Solve* (*conquer*) the subproblems
  – *Combine* the solutions to the subproblems to obtain a solution to the original problem

Merge Sort takes $\Theta(n \log n)$ time.
Review: Recurrence Relations

• Describe functions in terms of their values on smaller inputs

• Arise from Divide and Conquer

\[ T(n) = \Theta(1) \quad \text{if } n \leq c \]
\[ T(n) = a \ T(n/b) + D(n) + C(n) \quad \text{otherwise} \]

• Solution Methods (Chapter 4)
  – Substitution Method
  – Iteration Method
  – Master Method ■
Review: Substitution Method

• Guess the form of the solution, then use mathematical induction to show that it works

• Works well when the solution is easy to guess

• No general way to guess the correct solution
Review: Iteration Method

• Expand (iterate) the recurrence and express it as a summation of terms dependent only on $n$ and the initial conditions

• The key is to focus on 2 parameters
  – the number of times the recurrence needs to be iterated to reach the boundary condition
  – the sum of terms arising from each level of the iteration process

Review: Master Method

- Provides a “cookbook” method for solving recurrences of the form
  \[ T(n) = a \cdot T(n/b) + f(n) \]

- Assumptions:
  - \( a \geq 1 \) and \( b \geq 1 \) are constants
  - \( f(n) \) is an asymptotically positive function
  - \( T(n) \) is defined for nonnegative integers
  - We interpret \( n/b \) to mean either \( \lfloor n/b \rfloor \) or \( \lceil n/b \rceil \)
Outline

• Review
• **Recurrences (cont.)**
• The maximum subarray problem
• Elementary data structures
• Hash tables
• Binary search trees
The Master Theorem

- With the recurrence $T(n) = a \cdot T(n/b) + f(n)$ as in the previous slide, $T(n)$ can be bounded asymptotically as follows:

1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.

2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a \cdot \log n})$.

3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$, and if $a \cdot f(n/b) \leq c \cdot f(n)$ for some constant $c < 1$ and all sufficiently large $n$, then $T(n) = \Theta(f(n))$. ■
Simplified Master Theorem

Let $a \geq 1$ and $b > 1$ be constants and let $T(n)$ be the recurrence

$$T(n) = a \ T(n/b) + c \ n^k$$

defined for $n \geq 0$.

1. If $a > b^k$, then $T(n) = \Theta( n^{\log_b a} )$.

2. If $a = b^k$, then $T(n) = \Theta( n^k \lg n )$.

3. If $a < b^k$, then $T(n) = \Theta( n^k )$. ■
Proof of the master theorem

Figure 4.3 The recursion tree generated by $T(n) = aT(n/b) + f(n)$. The tree is a complete $a$-ary tree with $n^{\log_b a}$ leaves and height $\log_b n$. The cost of each level is shown at the right, and their sum is given in equation (4.6).
Proof of the master theorem

• Use blackboard
Examples

• $T(n) = 16T(n/4) + n$
  - $a = 16, b = 4$, thus $n^{\log_b a} = n^{\log_4 16} = \Theta(n^2)$
  - $f(n) = n = O(n^{\log_4 16 - \varepsilon})$ where $\varepsilon = 1 \Rightarrow \text{case 1}$.
  - Therefore, $T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$

• $T(n) = T(3n/7) + 1$
  - $a = 1, b=7/3$, and $n^{\log_b a} = n^{\log 7/3} = n^0 = 1$
  - $f(n) = 1 = \Theta(n^{\log_b a}) \Rightarrow \text{case 2}$.
  - Therefore, $T(n) = \Theta(n^{\log_b a} \log n) = \Theta(\log n)$ ■
Examples (Cont.)

• \( T(n) = 3T(n/4) + n \log n \)
  - \( a = 3, \ b=4, \) thus \( n^{\log_b a} = n^{\log_4 3} = O(n^{0.793}) \)
  - \( f(n) = n \log n = \Omega(n^{\log_4 3} + \varepsilon) \) where \( \varepsilon \approx 0.2 \Rightarrow \text{case 3}. \)
  - Therefore, \( T(n) = \Theta(f(n)) = \Theta(n \log n) \)

• \( T(n) = 2T(n/2) + n \log n \)
  - \( a = 2, \ b=2, \) \( f(n) = n \log n, \) and \( n^{\log_b a} = n^{\log_2 2} = n \)
  - \( f(n) \) is asymptotically larger than \( n^{\log_b a}, \) but not polynomially larger. The ratio \( \log n \) is asymptotically less than \( n^\varepsilon \) for any positive \( \varepsilon. \) Thus, the Master Theorem doesn’t apply here. ■
Outline

• Review
• Recurrences (cont.)
• **The maximum subarray problem**
• Elementary data structures
• Hash tables
• Binary search trees
The maximum subarray problem

• To invest in a company
  – Stock price goes up and down
  – Fig 4.1 (p. 68)
  – To maximize profit: buy low sell high

• A brute force solution \( C(n,2) = \Theta(n^2) \).
• A transformation: instead of looking at the daily prices, let us instead consider daily change in price.
  – Fig 4.3 (p. 70)
The maximum subarray problem

• The maximum subarray problem is the task of finding the contiguous subarray within a one-dimensional array of numbers (containing at least one positive number) which has the largest sum.
  – Justify the transformation: $a_j - a_i = \sum_{k=i}^{j} (a_{k+1} - a_k)$

• A solution using divide-and-conquer
  – pp. 70 - 74
The maximum subarray problem

- Use blackboard
Outline

• Review
• Recurrences (cont.)
• The maximum subarray problem
• Elementary data structures
• Hash tables
• Binary search trees
Dynamic Sets

• The rest of this lecture will focus on data structures rather than straight algorithms.
• Sets that grow, shrink, or otherwise change with time are called dynamic sets.
• Each element in the set is represented by a data object.
• Usually one of the fields of an object is called a key, and plays a central role in the manipulation of the set data.
Dynamic Sets

• Operations on a dynamic set generally fall into two categories,
  1. queries that return information about the set,
  2. modifying operations that change the set.
Dynamic Sets

• Dynamic sets support *queries* such as:
  
  • \text{Search}(S, k), \text{Minimum}(S), \text{Maximum}(S), \text{Successor}(S, x), \text{Predecessor}(S, x)

\text{SEARCH}(S,k). A query that given a set \( S \) and a key value \( k \) returns pointer \( x \) to element in \( S \) such that \( \text{key}[x]=k \), or NIL if no such element is found. ■
Dynamic Sets

• Dynamic sets support *queries* such as:
  - $\text{Search}(S, k)$, $\text{Minimum}(S)$, $\text{Maximum}(S)$,
    $\text{Successor}(S, x)$, $\text{Predecessor}(S, x)$

$\text{MINIMUM}(S)$. A query on a totally ordered set $S$ that returns the element of $S$ with smallest key.

$\text{MAXIMUM}(S)$. A query on a totally ordered set $S$ that returns the element of $S$ with largest key.
Dynamic Sets

- Dynamic sets support queries such as:
  - $\text{Search}(S, k)$, $\text{Minimum}(S)$, $\text{Maximum}(S)$, $\text{Successor}(S, x)$, $\text{Predecessor}(S, x)$

$\text{SUCCESSOR}(S,x)$. A query that, given an element $x$ whose key is from a totally ordered set, returns the next largest element in $S$, or NIL if there is no such element.

$\text{PREDECESSOR}(S,x)$. A query that, given an element $x$ whose key is from a totally ordered set, returns the next smallest element in $S$, or NIL if there is no such element.
Dynamic Sets

• Dynamic sets support *modifying operations* like:
  
  • Insert(S, x), Delete(S, x)

**INSERT(S,x).** A modifying operation that augments the set S with element pointed to by x.

**DELETE(S,x).** A modifying operation that given a pointer x to an element in the set S removes x from S.
Elementary Data Structures

• Stacks
• Queues
• Linked lists
• Rooted trees
Elementary Data Structures

• **Stacks**
  - LIFO (last-in first-out): The element deleted from the set is the one most recently inserted
  - We can simply use an array to implement a stack.
  - Underflow/Overflow
Elementary Data Structures

- Stack operations:
  Push (insert),
  Pop (delete)
  $O(1)$ time

```plaintext
STACK-EMPTY(S)
1 if top[S] = 0
2 then return TRUE
3 else return FALSE

PUSH(S, x)
1 top[S] ← top[S] + 1
2 S[top[S]] ← x

POP(S)
1 if STACK-EMPTY(S)
2 then error "underflow"
3 else top[S] ← top[S] − 1
4 return S[top[S] + 1]
```
Figure 10.1  An array implementation of a stack $S$. Stack elements appear only in the lightly shaded positions. (a) Stack $S$ has 4 elements. The top element is 9. (b) Stack $S$ after the calls \texttt{PUSH}(S, 17) and \texttt{PUSH}(S, 3). (c) Stack $S$ after the call \texttt{POP}(S) has returned the element 3, which is the one most recently pushed. Although element 3 still appears in the array, it is no longer in the stack; the top is element 17.
Elementary Data Structures

• **Queues**
  - FIFO (first-in first-out).
  - The queue has a head (where elements are dequeued) and a tail (enqueued).
  - We can simply use an array to implement a queue.
  - Underflow/Overflow
Elementary Data Structures

- Queue operations:
  - enqueue (insert),
  - dequeue (delete)

  $O(1)$ time

**ENQUEUE**($Q, x$)
1. $Q[tail(Q)] \leftarrow x$
2. if $tail(Q) = length(Q)$
3. then $tail(Q) \leftarrow 1$
4. else $tail(Q) \leftarrow tail(Q) + 1$

**DEQUEUE**($Q$)
1. $x \leftarrow Q[head(Q)]$
2. if $head(Q) = length(Q)$
3. then $head(Q) \leftarrow 1$
4. else $head(Q) \leftarrow head(Q) + 1$
5. return $x$
Figure 10.2  A queue implemented using an array $Q[1..12]$. Queue elements appear only in the lightly shaded positions. (a) The queue has 5 elements, in locations $Q[7..11]$. (b) The configuration of the queue after the calls \textsc{Enqueue}(Q, 17), \textsc{Enqueue}(Q, 3), and \textsc{Enqueue}(Q, 5). (c) The configuration of the queue after the call \textsc{Dequeue}(Q) returns the key value 15 formerly at the head of the queue. The new head has key 6.
Elementary Data Structures

• Lists
  - The objects are arranged in a linear order.
  - In an array, the linear order is determined by the array indices. In a list, the linear order is determined by a pointer (next/prev) in each object.
  - singly linked, **doubly linked**, circular lists
  - overflow/underflow?
Elementary Data Structures

• List operations:
  list-search,
  list-insert,
  list-delete

O(1)/O(n) time?

```
LIST-SEARCH(L, k)
1  x ← head[L]
2  while x ≠ NIL and key[x] ≠ k
3      do x ← next[x]
4  return x

LIST-INSERT(L, x)
1  next[x] ← head[L]
2  if head[L] ≠ NIL
3      then prev[head[L]] ← x
4  head[L] ← x
5  prev[x] ← NIL

LIST-DELETE(L, x)
1  if prev[x] ≠ NIL
2      then next[prev[x]] ← next[x]
3  else head[L] ← next[x]
4  if next[x] ≠ NIL
5      then prev[next[x]] ← prev[x]
```
Figure 10.3  (a) A doubly linked list \( L \) representing the dynamic set \{1, 4, 9, 16\}. Each element in the list is an object with fields for the key and pointers (shown by arrows) to the next and previous objects. The \textit{next} field of the tail and the \textit{prev} field of the head are \texttt{NIL}, indicated by a diagonal slash. The attribute \textit{head} points to the head. (b) Following the execution of \texttt{LIST-INSERT}(L, x), where \( key[x] = 25 \), the linked list has a new object with key 25 as the new head. This new object points to the old head with key 9. (c) The result of the subsequent call \texttt{LIST-DELETE}(L, x), where \( x \) points to the object with key 4.
Implementing pointers and objects

- Multiple-array representation of objects

![Multiple-array representation of objects](image1)

- Single-array representation of objects

![Single-array representation of objects](image2)
Allocating and freeing objects

• We can keep the free objects in a singly linked list
• Garbage collector
Figure 10.7 The effect of the ALLOCATE-OBJECT and FREE-OBJECT procedures. (a) The list of Figure 10.5 (lightly shaded) and a free list (heavily shaded). Arrows show the free-list structure. (b) The result of calling ALLOCATE-OBJECT() (which returns index 4), setting key[4] to 25, and calling LIST-INSERT(L, 4). The new free-list head is object 8, which had been next[4] on the free list. (c) After executing LIST-DELETE(L, 5), we call FREE-OBJECT(5). Object 5 becomes the new free-list head, with object 8 following it on the free list.
Representing rooted trees

- **Binary trees**: \( p[x], \) \( \text{left}[x], \) \( \text{right}[x] \)

- Rooted trees with unbounded branching

Figure 10.9: The representation of a binary tree \( T \). Each node \( x \) has the fields \( p[x] \) (top), \( \text{left}[x] \) (lower left), and \( \text{right}[x] \) (lower right). The key fields are not shown.

Figure 10.10: The left-child, right-sibling representation of a tree \( T \). Each node \( x \) has fields \( p[x] \) (top), \( \text{left-child}[x] \) (lower left), and \( \text{right-sibling}[x] \) (lower right). Keys are not shown.
Outline

• Review
• Recurrences (cont.)
• The maximum subarray problem
• Elementary data structures
• Hash tables
• Binary search trees
Hashing Tables

• Motivation: symbol tables
  – A compiler uses a symbol table to relate symbols to associated data
    • Symbols: variable names, procedure names, etc.
    • Associated data: memory location, call graph, etc.
  – For a symbol table (also called a dictionary), we care about search, insertion, and deletion
  – We typically don’t care about sorted order
Hash Tables

• More formally:
  – Given a table $T$ and a record $x$, with key (= symbol) and satellite data, we need to support:
    • Insert $(T, x)$
    • Delete $(T, x)$
    • Search$(T, x)$
  – We want these to be fast, but don’t care about sorting the records

• The structure we will use is a hash table
  – Supports all the above in $O(1)$ expected time! ■
Hashing: Keys

- In the following discussions we will consider all keys to be (possibly large) natural numbers
Direct Addressing

• Suppose:
  – The range of keys is 0..m-1
  – Keys are distinct

• The idea:
  – Set up an array T[0..m-1] in which
    • T[i] = x if x ∈ T and key[x] = i
    • T[i] = NULL otherwise
  – This is called a direct-address table
    • Operations take O(1) time!
      • *So what’s the problem?*
The Problem With Direct Addressing

- Direct addressing works well when the range $m$ of keys is relatively small
- But what if the keys are 32-bit integers?
  - **Problem 1**: direct-address table will have $2^{32}$ entries, more than 4 billion
  - **Problem 2**: even if memory is not an issue, the time to initialize the elements to NULL may be

- **Solution**: map keys to smaller range $0..m-1$
- This mapping is called a *hash function*
Hash Functions

- Next problem: collision

\[ h(k_1) = h(k_2) = h(k_5) \]

\[ U \quad (\text{universe of keys}) \]

\[ K \quad (\text{actual keys}) \]

\[ T \]

- 0
- \( h(k_1) \)
- \( h(k_4) \)
- \( h(k_2) = h(k_5) \)
- \( h(k_3) \)
- \( m - 1 \)
Resolving Collisions

• *How can we solve the problem of collisions?*
• Solution 1: *chaining*
• Solution 2: *open addressing*
Open Addressing

• Basic idea (details in Section 11.4):
  – To insert: if slot is full, try another slot, …, until an open slot is found (*probing*)
  – To search, follow same sequence of probes as would be used when inserting the element
    • If reach element with correct key, return it
    • If reach a NULL pointer, element is not in table

• Good for fixed sets (adding but no deletion, why?)
  – Example: spell checking

• Table needn’t be much bigger than $n$
Chaining

• Chaining puts elements that hash to the same slot in a linked list:
Chaining

• How do we insert an element?
Chaining

• How do we delete an element?
  – Do we need a doubly-linked list for efficient delete?
Chaining

- How do we search for an element with a given key?

$U$ (universe of keys)

$K$ (actual keys)

$T$
Collision resolution by chaining

• The dictionary operations on a hash table $T$ are easy to implement when collisions are resolved by chaining.
  – CHAINED-HASH-INSERT($T,x$)
    insert $x$ at the head of list $T[h(key[x])]$
  – CHAINED-HASH-SEARCH($T,k$)
    search for an element with key $k$ in list $T[h(k)]$
  – CHAINED-HASH-DELETE($T,x$)
    delete $x$ from the list $T[h(key[x])]$ ■
Collision resolution by chaining

- The worst-case running time for **insertion** is $O(1)$.
- For **searching**, the worst-case running time is proportional to the length of the list; we shall analyze this more closely below.
- **Deletion** of an element $x$ can be accomplished in $O(1)$ time if the lists are doubly linked.
  - If the lists are singly linked, we must first find $x$ in the list $T[h(key[x])]$, so that the *next* link of $x$'s predecessor can be properly set to splice $x$ out; in this case, deletion and searching have essentially the same running time.  $\square$
Analysis of Chaining

• Assume *simple uniform hashing*: each key in table is equally likely to be hashed to any slot

• Given $n$ keys and $m$ slots in the table: the *load factor* $\alpha = n/m = \text{average } \# \text{ keys per slot}$
Analysis of Chaining

• **What will be the average cost of an unsuccessful search for a key?**  
  A: $O(1 + \alpha)$ (intuitively, the expected length of the link list)

• **What will be the average cost of a successful search?**  
  A: $O(1 + \alpha/2)$ = $O(1 + \alpha)$ (intuitively, half the expected length of the link list)

• **Q: What is the difference between an unsuccessful search and a successful search?**
Analysis of Chaining

• **Theorem 11.1**
  In a hash table in which collisions are resolved by chaining, an *unsuccessful* search takes time $O(1 + \alpha)$, on the average, under the assumption of simple uniform hashing.

• **Proof:** *(Use blackboard)*
Analysis of Chaining

• **Theorem 11.2**
  In a hash table in which collisions are resolved by chaining, a *successful* search takes time $O(1 + \alpha)$, on the average, under the assumption of simple uniform hashing.

• **Proof:** *(Use blackboard)*
Analysis of Chaining Continued

• So the cost of searching $= O(1 + \alpha)$

• *If the number of hash-table slots is at least proportional to the number of elements in the table, we have $n = O(m)$.*
  
  $\alpha = n/m = O(m)/m = O(1)$
  
  – In other words, we can make the expected cost of searching constant if we make $\alpha$ constant.
Choosing A Hash Function

• Clearly choosing the hash function well is crucial
  – What will a worst-case hash function do?
  – What will be the time to search in this case?
• What are desirable features of the hash function?
  – Should distribute keys uniformly into slots
  – Should not depend on patterns in the data
Hash Functions: The Division Method

• \( h(k) = k \mod m \)
  – In words: hash \( k \) into a table with \( m \) slots using the slot given by the remainder of \( k \) divided by \( m \)

• What happens to elements with adjacent values of \( k \)?
• What happens if \( m \) is a power of 2 (say \( 2^P \))?
• What if \( m \) is a power of 10?

• It is better to make the hash function depend on all the bits (or digits) of the key
  – Pick table size \( m = \) prime number not too close to a power of 2 (or 10)
Hash Functions: The Multiplication Method

- For a constant $A$, $0 < A < 1$:
- $h(k) = \lfloor m (kA - \lfloor kA \rfloor) \rfloor$

What does this term represent?
Hash Functions:
The Multiplication Method

• For a constant $A$, $0 < A < 1$:
• $h(k) = \lfloor m \times (kA - \lfloor kA \rfloor) \rfloor$

  \[\text{Fractional part of } kA\]

• Choose $m = 2^p$
• Choose $A$ not too close to 0 or 1
• Knuth: Good choice for $A = (\sqrt{5} - 1)/2$
Hash Functions: Worst Case Scenario

- It is always possible to analyzes a hash function and pick a sequence of “worst-case” keys that all hash to the same slot, yielding an average retrieval time $\Theta(n)$.
- What’s can we do?
Hash Functions: Universal Hashing

• To foil such malicious adversaries: randomize the algorithm
  
  • *Universal hashing*: pick a hash function randomly in a way that is independent of the keys that are actually going to be stored
    – Guarantees good performance on average, no matter what keys adversary chooses
    – *Details in 11.3.3* ■
Outline

• Review
• Recurrences (cont.)
• The maximum subarray problem
• Elementary data structures
• Hash tables
• **Binary search trees**
Binary Search Trees

• *Binary Search Trees* (BSTs) are binary trees with **binary search tree property**:

\[
key[leftSubtree(x)] \leq key[x] \leq key[rightSubtree(x)]
\]

• In addition to satellite data, elements in a BST have:
  – *key*: an identifying field inducing a total ordering
  – *left*: pointer to a left child (may be NULL)
  – *right*: pointer to a right child (may be NULL)
  – *p*: pointer to a parent node (NULL for root) ■
Binary Search Trees

• Examples:

![Binary Search Tree Diagrams](image)

Figure 12.1  Binary search trees. For any node \( x \), the keys in the left subtree of \( x \) are at most \( \text{key}[x] \), and the keys in the right subtree of \( x \) are at least \( \text{key}[x] \). Different binary search trees can represent the same set of values. The worst-case running time for most search-tree operations is proportional to the height of the tree. (a) A binary search tree on 6 nodes with height 2. (b) A less efficient binary search tree with height 4 that contains the same keys.
Inorder Tree Walk

• What does the following code do?
  
  TreeWalk(x)
  
  TreeWalk(left[x]);
  
  print(x);
  
  TreeWalk(right[x]);

• A: prints elements in sorted (increasing) order
• This is called an *inorder tree walk*
  
  – *Preorder tree walk*: print root, then left, then right
  
  – *Postorder tree walk*: print left, then right, then root
Inorder Tree Walk

• Example:

• How long will a tree walk take? **What is the recurrence** $T(n)$?

• Prove that inorder walk prints in monotonically increasing order.
Operations on BSTs: Search

- Given a key $k$ and a pointer to a node $x$, returns an element with that key or NULL:

```plaintext
TREE-SEARCH(x, k)
1    if x = NIL or k = key[x]
2      then return x
3    if k < key[x]
4      then return TREE-SEARCH(left[x], k)
5  else return TREE-SEARCH(right[x], k)

ITERATIVE-TREE-SEARCH(x, k)
1    while x ≠ NIL and k ≠ key[x]
2      do if k < key[x]
3        then x ← left[x]
4      else x ← right[x]
5    return x
```
Operations on BSTs: Search

- Example:

Figure 12.2 Queries on a binary search tree. To search for the key 13 in the tree, we follow the path 15 → 6 → 7 → 13 from the root. The minimum key in the tree is 2, which can be found by following left pointers from the root. The maximum key 20 is found by following right pointers from the root. The successor of the node with key 15 is the node with key 17, since it is the minimum key in the right subtree of 15. The node with key 13 has no right subtree, and thus its successor is its lowest ancestor whose left child is also an ancestor. In this case, the node with key 15 is its successor.
Operations on BSTs: Search

• *How long will a search operation take?*
• $T(n) = ?$

```
TREE-SEARCH(x, k)
1    if x = NIL or k = key[x]
2        then return x
3    if k < key[x]
4        then return TREE-SEARCH(left[x], k)
5    else return TREE-SEARCH(right[x], k)
```
Operations of BSTs: Insert

• Adds an element $z$ to the tree so that the binary search tree property continues to hold

• The basic algorithm
  – Like the search procedure above
  – Insert $z$ in place of NULL
TREE-INSERT($T$, $z$)
1   $y \leftarrow$ NIL
2   $x \leftarrow root[T]$
3   while $x \neq$ NIL
4       do $y \leftarrow x$
5              if $key[z] < key[x]$
6                  then $x \leftarrow left[x]$
7              else $x \leftarrow right[x]$
8   $p[z] \leftarrow y$
9   if $y =$ NIL
10      then $root[T] \leftarrow z$  // Tree $T$ was empty
11 else if $key[z] < key[y]$
12      then $left[y] \leftarrow z$
13 else $right[y] \leftarrow z$
BST Insert: Example

- Example: Insert C
BST Search/Insert: Running Time

• What is the running time of TreeSearch( ) or TreeInsert( )?
• A: $O(h)$, where $h = \text{height of tree}$
• What is the height of a binary search tree?
• A: worst case: $h = O(n)$ when tree is just a linear string of left or right children
  – We’ll keep all analysis in terms of $h$ for now
  – Later we’ll see how to maintain $h = O(\lg n)$
Sorting again

- **Sorting With Binary Search Trees**
  Informal code for sorting array A of length $n$:
  
  ```plaintext
  BSTSort(A)
  for i=1 to n
    TreeInsert(A[i]);
  InorderTreeWalk(root);
  ```

- Argue that this is $\Omega(n \log n)$
- What will be the running time in the
  - Worst case?
  - Average case?
Sorting With BSTs

for i=1 to n
  TreeInsert(A[i]);
  InorderTreeWalk(root);
Moe BST Operations: Minimum, Maximum

• How can we implement Minimum and Maximum queries?

• What are their running time?

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

TREE-MAXIMUM(x)
1 while right[x] ≠ NIL
2 do x ← right[x]
3 return x
```
More BST Operations: Successor and Predecessor

• For deletion, we will need a Successor() operation.

• What are the general rules for finding the successor of node x?

• Two cases:
  – x has a right subtree: successor is minimum node in right subtree
  – x has no right subtree: successor is first ancestor of x whose left child is also ancestor of x
    • Intuition: As long as you move to the left up the tree, you’re visiting smaller nodes.

• Predecessor: similar algorithm

```
TreeWalk(x)
TreeWalk(left[x]);
print(x);
TreeWalk(right[x]);
```
BST Operations: Successor

TREE-SUCCESSOR \( x \)
1. if \( \text{right}[x] \neq \text{NIL} \)
2. then return TREE-MINIMUM(\( \text{right}[x] \))
3. \( y \leftarrow p[x] \)
4. while \( y \neq \text{NIL} \) and \( x = \text{right}[y] \)
5. do \( x \leftarrow y \)
6. \( y \leftarrow p[y] \)
7. return \( y \)
BST Operations: Successor

• *Example:*

*What is the successor of node 3? Node 15? Node 13?*
BST Operations: Delete

• Deletion is a bit tricky
• 3 cases:
  – (0). x has no children:
    • Remove x
  – (1). x has one child:
    • Splice out x
  – (2). x has two children:
    • Swap x with its successor y
    • Perform case (0) or (1) to delete y
BST Operations: Delete

• Why will case 2 always go to case 0 or case 1?
• A: because when x has 2 children, its successor is the minimum in its right subtree
• Could we swap x with predecessor instead of successor?
• A: yes.
TREE-DELETE \((T, z)\)

1. \(\text{if } \text{left}[z] = \text{NIL} \text{ or } \text{right}[z] = \text{NIL} \)
2. \(\text{then } y \leftarrow z\)
3. \(\text{else } y \leftarrow \text{TREE-SUCCESSOR}(z)\)
4. \(\text{if } \text{left}[y] \neq \text{NIL} \)
5. \(\text{then } x \leftarrow \text{left}[y]\)
6. \(\text{else } x \leftarrow \text{right}[y]\)
7. \(\text{if } x \neq \text{NIL} \)
8. \(\text{then } p[x] \leftarrow p[y]\)
9. \(\text{if } p[y] = \text{NIL} \)
10. \(\text{then } \text{root}[T] \leftarrow x\)
11. \(\text{else if } y = \text{left}[p[y]]\)
12. \(\text{then } \text{left}[p[y]] \leftarrow x\)
13. \(\text{else } \text{right}[p[y]] \leftarrow x\)
14. \(\text{if } y \neq z\)
15. \(\text{then } \text{key}[z] \leftarrow \text{key}[y]\)
16. \(\text{copy } y's \text{ satellite data into } z\)
17. \(\text{return } y\)
**Figure 12.4** Deleting a node $z$ from a binary search tree. Which node is actually removed depends on how many children $z$ has; this node is shown lightly shaded. (a) If $z$ has no children, we just remove it. (b) If $z$ has only one child, we splice out $z$. (c) If $z$ has two children, we splice out its successor $y$, which has at most one child, and then replace $z$’s key and satellite data with $y$’s key and satellite data.
Binary search trees: conclusion

• Search trees are data structures that support many dynamic-set operations.

• Basic operations on a binary search tree take time propositional to the \textit{height} of the tree.
  – The \textit{height of a node} in a tree is the number of edges on the longest simple downward path from the node to a leaf.
  – The \textit{height of a tree} is the height of its root.

• Up next: guaranteeing a $O(\log n)$ height tree