Chapter 3 (Data Structures)

1. Role of Data Structures
2. Review: Contiguous vs. Linked Data Structures
3. Review: Stacks & Queues
4. Review: Dictionaries
5. Review: Hash Tables
6. Binary Search Trees
7. Balanced Binary Search Trees

Thoughts for the Day

"Mankind's progress is measured by the number of things we can do without thinking."

Thoughts for the Day

"Mankind's progress is measured by the number of things we can do without thinking."

"Changing a data structure in a slow program can work the same way an organ transplant does in a sick patient"

Elementary Data Structures

Elementary data structures such as stacks, queues, lists, and heaps are the "off-the-shelf" components we build our algorithm from.

Abstraction v/s Implementation

There are two aspects to any data structure:

- The abstract operations which it supports.
- The implementation of these operations.

Data Abstraction

- That we can describe the behavior of our data structures in terms of abstract operations is why we can use them without thinking.

Data Abstraction

- That we can describe the behavior of our data structures in terms of abstract operations is why we can use them without thinking.
- That there are different implementations of the same abstract operations enables us to optimize performance.

Contiguous vs. Linked Data Structures

Data structures can be neatly classified as either contiguous or linked depending upon whether they are based on arrays or pointers:

Contiguous vs. Linked Data Structures

Data structures can be neatly classified as either contiguous or linked depending upon whether they are based on arrays or pointers:

- Contiguously-allocated structures are composed of single slabs of memory, and include arrays, matrices, heaps, and hash tables.

Contiguous vs. Linked Data Structures

Data structures can be neatly classified as either contiguous or linked depending upon whether they are based on arrays or pointers:

- Contiguously-allocated structures are composed of single slabs of memory, and include arrays, matrices, heaps, and hash tables.
- Linked data structures are composed of multiple distinct chunks of memory bound together by *pointers*, and include lists, trees, and graph adjacency lists.

Arrays

An array is a structure of fixed-size data records such that each element can be efficiently located by its index or (equivalently) address.

Advantages of contiguously-allocated arrays

Advantages of contiguously-allocated arrays

- Constant-time ($\Theta(1)$) access given the index.

Advantages of contiguously-allocated arrays

- Constant-time ($\Theta(1)$) access given the index.
- Arrays consist purely of data, so no space is wasted with links or other formatting information.

Advantages of contiguously-allocated arrays

- Constant-time ($\Theta(1)$) access given the index.
- Arrays consist purely of data, so no space is wasted with links or other formatting information.
- Physical continuity (memory locality) between successive data accesses helps exploit the high-speed cache memory on modern computer architectures.

Dynamic Arrays

- Unfortunately we cannot adjust the size of simple arrays in the middle of a program's execution.
- Compensating by allocating extremely large arrays can waste a lot of space.
- With *dynamic arrays* we start with an array of size 1, and double its size each time we run out of space.
- How many times will we double for n elements?

Dynamic Arrays

- Unfortunately we cannot adjust the size of simple arrays in the middle of a program's execution.
- Compensating by allocating extremely large arrays can waste a lot of space.
- With *dynamic arrays* we start with an array of size 1, and double its size each time we run out of space.
- How many times will we double for n elements? Answer: $\lceil \log_2 n \rceil$.

Dynamic Array Worksheet

How Much Total Work?

The apparent waste involves the recopying of old contents on each expansion.

How Much Total Work?

The apparent waste involves the recopying of old contents on each expansion.
If half the elements move once, a quarter of the elements twice, and so on, the total number of movements M is given by .....

Analysis

$$M = \sum_{i=1}^{\lg n} i \cdot n/2^i \quad = n \sum_{i=1}^{\lg n} i/2^i$$

Analysis

$$M = \sum_{i=1}^{\lg n} i \cdot n/2^i = n \sum_{i=1}^{\lg n} i/2^i$$

$$\leq n \sum_{i=1}^{\infty} i/2^i = 2n$$

Analysis

$$M = \sum_{i=1}^{\lg n} i \cdot n/2^i = n \sum_{i=1}^{\lg n} i/2^i$$

$$\leq n \sum_{i=1}^{\infty} i/2^i = 2n$$

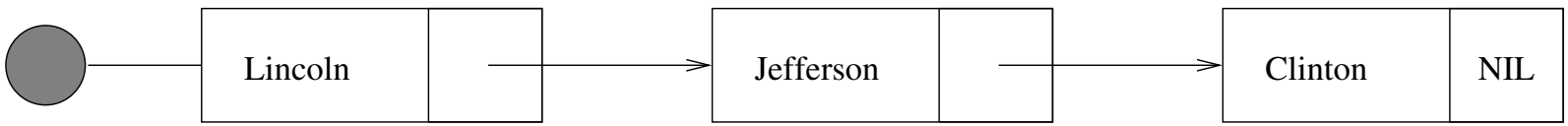
Thus each of the n elements move an average of only twice, and the total work of managing the dynamic array is $O(n)$ (same as a simple array)!

Pointers and Linked Structures

- Pointers represent the address of a location in memory.
- A cell-phone number can be thought of as a pointer to its owner as they move about the planet.
- In C/C++, `*p` denotes the item pointed to by `p`, and `&x` denotes the address (i.e. pointer) of a particular variable `x`.
- A special `NULL` pointer value is used to denote structure-terminating or unassigned pointers.

Linked List Structures

```
typedef struct list {
        item_type item;
        struct list *next;
} list;
```



Operations on Lists

1. Search: walk thru list until you find x.

[1] story

Operations on Lists

1. Search: walk thru list until you find x. $O(n)$.

[1] story

Operations on Lists

1. Search: walk thru list until you find x. $O(n)$.
2. Insert: insert at front [1].

[1] story

Operations on Lists

1. Search: walk thru list until you find x. $O(n)$.
2. Insert: insert at front [1]. $\Theta(1)$

[1] story

Operations on Lists

1. Search: walk thru list until you find x. $O(n)$.
2. Insert: insert at front [1]. $\Theta(1)$
3. Delete: first search for x, then delete.

[1] story

Operations on Lists

1. Search: walk thru list until you find x. $O(n)$.
2. Insert: insert at front [1]. $\Theta(1)$
3. Delete: first search for x, then delete. $O(n)$

Why $O(n)$ and $\Theta(1)$ above??

[1] story

Advantages of Linked Lists

The relative advantages of linked lists over static arrays include:

Advantages of Linked Lists

The relative advantages of linked lists over static arrays include:

1. Overflow on linked structures can never occur unless the memory is actually full.

Advantages of Linked Lists

The relative advantages of linked lists over static arrays include:

1. Overflow on linked structures can never occur unless the memory is actually full.
2. Insertions and deletions are *simpler* than for contiguous (array) lists.

Advantages of Linked Lists

The relative advantages of linked lists over static arrays include:

1. Overflow on linked structures can never occur unless the memory is actually full.
2. Insertions and deletions are *simpler* than for contiguous (array) lists.
3. With large records, moving pointers is easier and faster than moving the items themselves.

Advantages of Linked Lists

The relative advantages of linked lists over static arrays include:

1. Overflow on linked structures can never occur unless the memory is actually full.
2. Insertions and deletions are *simpler* than for contiguous (array) lists.
3. With large records, moving pointers is easier and faster than moving the items themselves.

Dynamic memory allocation provides us with flexibility on how and where we use our limited storage resources.

Comic Break (pointers)

Stacks and Queues

Sometimes, the order in which we retrieve data is independent of its content, being only a function of when it arrived.

Stacks and Queues

Sometimes, the order in which we retrieve data is independent of its content, being only a function of when it arrived.

A stack supports last-in, first-out operations: push and pop.

A queue supports first-in, first-out operations: enqueue and dequeue.

Stacks and Queues

Sometimes, the order in which we retrieve data is independent of its content, being only a function of when it arrived.

A stack supports last-in, first-out operations: push and pop.

A queue supports first-in, first-out operations: enqueue and dequeue.

A deque is a double ended queue and supports all four operations: push, pop, enqueue, dequeue.

Stacks and Queues

Sometimes, the order in which we retrieve data is independent of its content, being only a function of when it arrived.

A stack supports last-in, first-out operations: push and pop.

A queue supports first-in, first-out operations: enqueue and dequeue.

A deque is a double ended queue and supports all four operations: push, pop, enqueue, dequeue.

Lines in banks are based on queues, while a pile of dishes is a stack.

Dictionary / Dynamic Set

Dictionary: set of items indexed by keys.

Dictionary / Dynamic Set

Dictionary: set of items indexed by keys.

- *Search(S,k)* – Given a set S and a key value k, returns a ptr x to an element in S whose key is k, if it exists.
- *Insert(S,x)* – Add x to set S.
- *Delete(S,x)* – Given a pointer x to an element in the set S, remove x from S. (Observe we are given a ptr, not a key value).

Dictionary / Dynamic Set

Dictionary: set of items indexed by keys.

- *Search(S,k)* – Given a set S and a key value k, returns a ptr x to an element in S whose key is k, if it exists.
- *Insert(S,x)* – Add x to set S.
- *Delete(S,x)* – Given a pointer x to an element in the set S, remove x from S. (Observe we are given a ptr, not a key value).

Above are primary operations.

Auxiliary Operations (need ordered sets)

- *Min(S), Max(S)* – Returns the element of the totally ordered set S which has the smallest (largest) key.
- *Successor(S,x), Predecessor(S,x)* – Given an element x whose key is from a totally ordered set S, returns the next largest (smallest) element in S, or NIL if x is the maximum (minimum) element.

Auxiliary Operations (need ordered sets)

- *Min(S), Max(S)* – Returns the element of the totally ordered set S which has the smallest (largest) key.
- *Successor(S,x), Predecessor(S,x)* – Given an element x whose key is from a totally ordered set S, returns the next largest (smallest) element in S, or NIL if x is the maximum (minimum) element.
  aka Next/Previous

Dictionary Implementations

1. Unsorted Array
2. Sorted Array
3. Singly-Linked Unsorted List
4. Doubly-Linked Unsorted List
5. Singly-Linked Sorted List
6. Doubly-Linked Sorted List
7. Binary Search Tree
8. Balanced Binary Search Tree
9. Hash Tables

Array Based Sets: Unsorted Arrays

In this context, ptr $=$ index

Array Based Sets: Unsorted Arrays

In this context, ptr $=$ index

- Search(S,k) - sequential search, $O(n)$

Array Based Sets: Unsorted Arrays

In this context, ptr $=$ index

- Search(S,k) - sequential search, $O(n)$
- Insert(S,x) - place in first empty spot (like dynamic array), $\Theta(1)$

Array Based Sets: Unsorted Arrays

In this context, ptr = index

- Search(S,k) - sequential search, $O(n)$
- Insert(S,x) - place in first empty spot (like dynamic array), $\Theta(1)$
- Delete(S,x) - copy nth item to the xth spot, $\Theta(1)$

Array Based Sets: Unsorted Arrays

In this context, ptr = index

- Search(S,k) - sequential search, $O(n)$
- Insert(S,x) - place in first empty spot (like dynamic array), $\Theta(1)$
- Delete(S,x) - copy nth item to the xth spot, $\Theta(1)$
- Min(S), Max(S) - sequential search, $\Theta(n)$

Array Based Sets: Unsorted Arrays

In this context, ptr = index

- Search(S,k) - sequential search, $O(n)$
- Insert(S,x) - place in first empty spot (like dynamic array), $\Theta(1)$
- Delete(S,x) - copy nth item to the xth spot, $\Theta(1)$
- Min(S), Max(S) - sequential search, $\Theta(n)$
- Successor(S,x), Predecessor(S,x) - sequential search, $\Theta(n)$

Array Based Sets: Sorted Arrays

Array Based Sets: Sorted Arrays

- Search(S,k) - binary search, $O(\lg n)$

Array Based Sets: Sorted Arrays

- Search(S,k) - binary search, $O(\lg n)$
- Insert(S,x) - search, then move to make space, $O(n)$

Array Based Sets: Sorted Arrays

- Search(S,k) - binary search, $O(\lg n)$
- Insert(S,x) - search, then move to make space, $O(n)$
- Delete(S,x) - move to fill up the hole, $O(n)$

Array Based Sets: Sorted Arrays

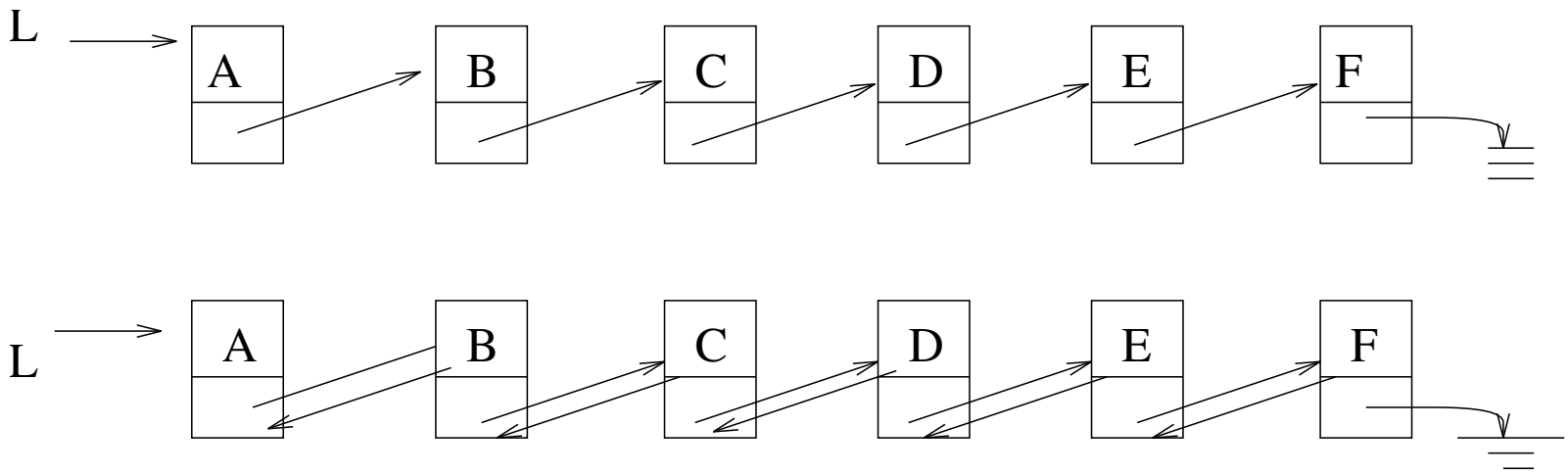
- Search(S,k) - binary search, $O(\lg n)$
- Insert(S,x) - search, then move to make space, $O(n)$
- Delete(S,x) - move to fill up the hole, $O(n)$
- Min(S), Max(S) - first or last element, $\Theta(1)$

Array Based Sets: Sorted Arrays

- Search(S,k) - binary search, $O(\lg n)$
- Insert(S,x) - search, then move to make space, $O(n)$
- Delete(S,x) - move to fill up the hole, $O(n)$
- Min(S), Max(S) - first or last element, $\Theta(1)$
- Successor(S,x), Predecessor(S,x) - Add or subtract 1 from pointer, $\Theta(1)$

Pointer Based Implementation

We can maintain a dictionary in either a singly or doubly linked list.



Doubly Linked Lists

- We gain extra flexibility on predecessor queries at a cost of doubling the number of pointers by using doubly-linked lists.
- The extra big-Oh costs of doubly-linked lists is zero.

Linked List Complexities

Op	Singly unsorted	Double unsorted	Singly sorted	Doubly sorted
Search	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Insert	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$
Delete	$O(n)$*	$\Theta(1)$	$O(n)$*	$\Theta(1)$
Succ/Pred	$O(n)$	$O(n)$	$\Theta(1)/O(n)$	$\Theta(1)$
Min/Max	$O(n)$	$O(n)$	$\Theta(1)$	$\Theta(1)$

Array Complexities (from earlier)

Op	Unsorted	Sorted
Search	$O(n)$	$O(\log n)$
Insert	$\Theta(1)$	$O(n)$
Delete	$\Theta(1)$	$O(n)$
Succ/Pred	$O(n)$	$\Theta(1)$
Min/Max	$O(n)$	$\Theta(1)$

DS Operation Expectations: Discussion

What complexity is considered reasonable for a data structure operation and why?

- $\Theta(1)$
- $O(n)$
- $O(\log n)$

Binary Search Trees

Binary search trees support all dictionary ops.

Op	Binary Search tree
Search	$O(h)$
Insert	$O(h)$
Delete	$O(h)$
Succ/Pred	$O(h)$
Min/Max	$O(h)$

h denotes the height of the tree and is $\Theta(\log n)$ on the average and $\Theta(n)$ in the worst case.

Representation of BST

- Uses pointers.
- Node contains
  - left & right (child) ptrs.
  - key field.
  - parent ptr.
- The parent link is optional, since we can store the ptrs on a stack on the way down.

Definition of BST

- Rooted[2] binary tree.
- Let x be any node in the BST.
- If y is a node in x's left subtree, then $y \rightarrow key \leq x \rightarrow key$.
- If y is a node in x's right subtree, then $y \rightarrow key \geq x \rightarrow key$.

[2] what's an unrooted tree?

Inorder Traversal of a BST

An inorder traversal of a BST prints all the keys in sorted order.

INORDER(x)

if ($x \neq 0$)

INORDER($x \rightarrow left$);

cout $<<$ $x \rightarrow key$;

INORDER($x \rightarrow right$);

Searching a BST

SEARCH(x,k)

if $(x == 0$ **or** $k == x \rightarrow key)$
then return x;
if $(k < x \rightarrow key)$
then return SEARCH$(x \rightarrow left, k)$;
else return SEARCH$(x \rightarrow right, k)$;

Searching in a BST: how much?

The algorithm works because both the left and right subtrees of a binary search tree *are* binary search trees – recursive structure, recursive algorithm. This takes time proportional to the height of the tree, $O(h)$.

Min/Max

- Where would you expect to find the minimum element? Left subtree, root, or right subtree?

Min/Max

- Where would you expect to find the minimum element? Left subtree, root, or right subtree?
- Answer: Left subtree. Need to follow left pointers:

while $(x \rightarrow left! = 0)$
$x = x \rightarrow left$
return x;

Min/Max

- Where would you expect to find the minimum element? Left subtree, root, or right subtree?
- Answer: Left subtree. Need to follow left pointers:

  while $(x \rightarrow left! = 0)$
  $x = x \rightarrow left$
  return x;

- Max is similar.

Min/Max

- Where would you expect to find the minimum element? Left subtree, root, or right subtree?
- Answer: Left subtree. Need to follow left pointers:

  while $(x \rightarrow left! = 0)$
  $x = x \rightarrow left$
  return x;

- Max is similar.
- Time Complexity: $O(h)$.

Successor (Predecessor is symmetric)

- Recall Successor $=$ the smallest element $> x$.

Successor (Predecessor is symmetric)

- Recall Successor $=$ the smallest element $> x$.
- Do you see a connection with INORDER?

Successor (Predecessor is symmetric)

- Recall Successor $=$ the smallest element $> x$.
- Do you see a connection with INORDER?
- Where is the successor?

Successor (Predecessor is symmetric)

- Recall Successor = the smallest element $> x$.
- Do you see a connection with INORDER?
- Where is the successor?
- If x has a right subtree, successor would be the smallest element in it.
- Else it's the nearest ancestor a(of x) such that x is in a's left subtree.

Successor (Predecessor is symmetric)

- Recall Successor = the smallest element $> x$.
- Do you see a connection with INORDER?
- Where is the successor?
- If x has a right subtree, successor would be the smallest element in it.
- Else it's the nearest ancestor a(of x) such that x is in a's left subtree.
- How do you get to an ancestor of a node?

Successor (Predecessor is symmetric)

- Recall Successor = the smallest element $> x$.
- Do you see a connection with INORDER?
- Where is the successor?
- If x has a right subtree, successor would be the smallest element in it.
- Else it's the nearest ancestor a(of x) such that x is in a's left subtree.
- How do you get to an ancestor of a node?
- $O(h)$ complexity.

Insertion

- Do a binary tree search to find where it should be, then replace the termination NIL ptr with the new item.
- Need to maintain a trailing pointer in the code.
- Insertion takes time proportional to the height of the tree, $O(h)$.

Insertion Code

INSERT(T,z)

$y = 0;$
$x = T.root;$
while $(x! = 0)$ **do**
$y = x$
if $(z \rightarrow key < x \rightarrow key)$
then $x = x \rightarrow left$
else $x = x \rightarrow right$

Insertion Code continued

$z \rightarrow p = y$
if $(y == 0)$ $T.root = z;$
elseif $(z \rightarrow key < y \rightarrow key)$
then $y \rightarrow left = z;$
else $y \rightarrow right = z;$

Deletion

1. First, search for the key z to be deleted from the BST.

Deletion

1. First, search for the key z to be deleted from the BST.
2. Three cases:
    1. z has no children: just remove z (and you're done).
    2. z has one child: splice z out (and you're done).
    3. z has two children. (Bit harder)

Deletion

1. First, search for the key z to be deleted from the BST.
2. Three cases:
    1. z has no children: just remove z (and you're done).
    2. z has one child: splice z out (and you're done).
    3. z has two children. (Bit harder)
        1. Let y be z's successor.
        2. Replace z's contents with y' contents.
        3. Splice y out (y has at most one child. Why?).

BST Runtime

All of our dictionary operations, when implemented with binary search trees, take $O(h)$, where h is the height of the tree.

BST Runtime

All of our dictionary operations, when implemented with binary search trees, take $O(h)$, where h is the height of the tree.
The best height we could hope to get is $\lg n$, if the tree was perfectly balanced, since

$$\sum_{i=0}^{\lfloor \lg n \rfloor} 2^i \approx n$$

BST Runtime

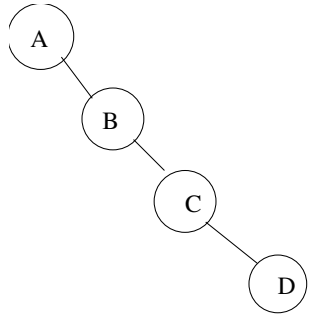
All of our dictionary operations, when implemented with binary search trees, take $O(h)$, where h is the height of the tree.
The best height we could hope to get is $\lg n$, if the tree was perfectly balanced, since

$$\sum_{i=0}^{\lfloor \lg n \rfloor} 2^i \approx n$$

But if we get unlucky with our order of insertion or deletion, we could get linear height!

Worst Case Height

insert(A); insert(B)
insert(C); insert(D)
$\vdots$

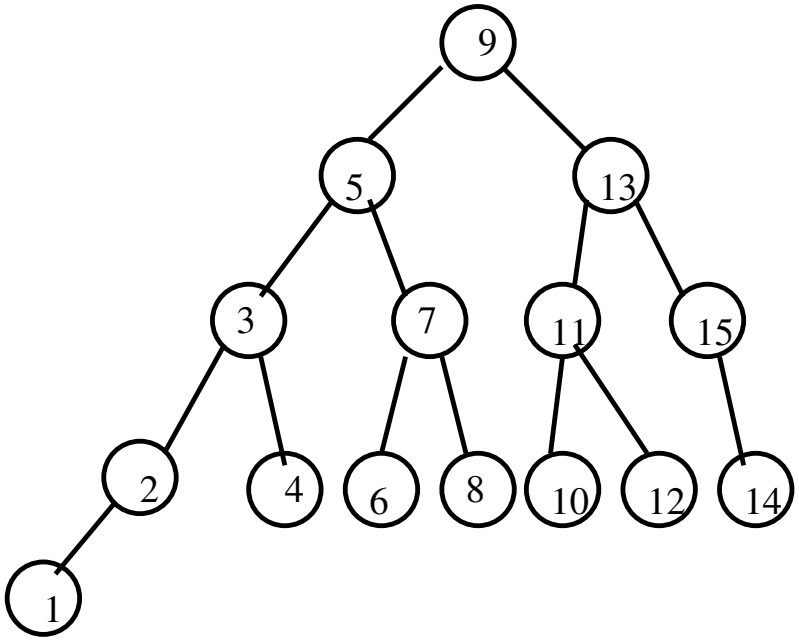


Tree Insertion Analysis

In fact, binary search trees constructed with random insertion orders *on average* have $\Theta(\lg n)$ height.
The worst case is linear, however.

Perfectly Balanced Trees

Perfectly balanced trees are high-maintenance: if we insert the key 1, we must move every single node in the tree to rebalance it, taking $\Theta(n)$ time.



Balanced Search Trees

Therefore, when we talk about "balanced" trees, we mean trees whose height is $O(\log n)$, so all dictionary operations (insert, delete, search, min/max, successor/predecessor) take $O(\lg n)$ time in the worst case.

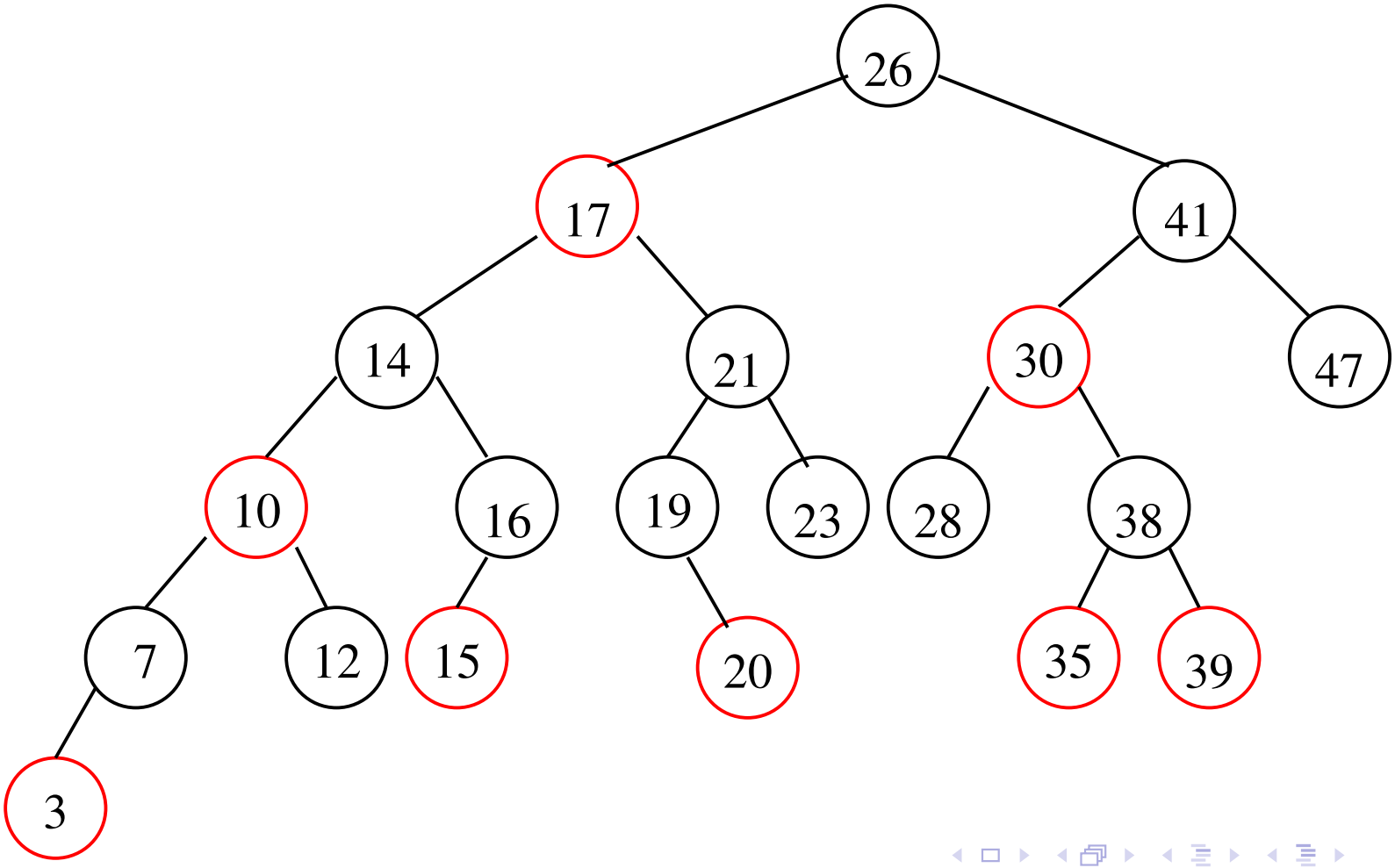
Red-Black Trees not in book

A bst is a red-black tree if it satisfies the following properties.
(1) Every node is either red or black.
(2) If a node is red, then both of its children are black.
(3) Every path from a node to a leaf contains the same number of black nodes.
(4) The root is always black.

Theorem

An RB tree with n nodes has $h \leq 2\log_2(n+1)$.

Example



Red-Black Tree Operations

Search, Min, Max, Predecessor, Successor are unchanged because they don't change the tree.

Red-Black Tree Operations

Search, Min, Max, Predecessor, Successor are unchanged because they don't change the tree. But, Insert and Delete will need to be modified so that they don't violate red-black tree properties.

Red-Black Tree Operations

Search, Min, Max, Predecessor, Successor are unchanged because they don't change the tree. But, Insert and Delete will need to be modified so that they don't violate red-black tree properties. Do this using rotations and color changes.

Rotation Worksheet

- Goal: understand rotation template.

Demo

- Goal: to see how red black trees self-adjust during insertion.

Skip Lists, Splay Trees

- Paris Kanellakis award
- Data Structures Handbook (Page 15, 173 (dictionary structures), 195 (balanced search trees), 235 (splay trees), 305 (B Trees)).

Hash Tables

- Hash tables are a *very practical* way to maintain a dictionary.

Hash Tables

- Hash tables are a *very practical* way to maintain a dictionary.
- The idea is simply that looking an item up in an array is $\Theta(1)$ once you have its index.

Hash Tables

- Hash tables are a *very practical* way to maintain a dictionary.
- The idea is simply that looking an item up in an array is $\Theta(1)$ once you have its index.
- A hash function is a mathematical function which maps keys to integers.

Hash Functions

Again, hash function's job is to map keys to integers.

Hash Functions

Again, hash function's job is to map keys to integers.
Ideal hash function:

1. Is cheap to evaluate
2. Element is equally likely to hash into any of the m slots.
3. independently of other elements.

Hash Functions

The first step is usually to map the key to a big integer; for example

$$h(k) = \sum_{i=0}^{keylength} 128^i \times char(key[i])$$

Hash Functions

The first step is usually to map the key to a big integer; for example

$$h(k) = \sum_{i=0}^{keylength} 128^i \times char(key[i])$$

This large number must be reduced to an integer whose value is between 0 and $m - 1$ (size of hash table).

Hash Functions

The first step is usually to map the key to a big integer; for example

$$h(k) = \sum_{i=0}^{keylength} 128^i \times char(key[i])$$

This large number must be reduced to an integer whose value is between 0 and $m - 1$ (size of hash table).
One way is by $h(k) = k \mod m$

The Birthday Paradox

No matter how good our hash function is, we'd better be prepared for collisions because of the birthday paradox.

The Birthday Paradox

No matter how good our hash function is, we'd better be prepared for collisions because of the birthday paradox.
The probability of there being no collisions after n insertions into an m-element table is

The Birthday Paradox

No matter how good our hash function is, we'd better be prepared for collisions because of the birthday paradox.

The probability of there being no collisions after n insertions into an m-element table is

$$\frac{m}{m} \times \frac{m-1}{m} \times \ldots \times \frac{m-n+1}{m} = \prod_{i=0}^{n-1} \frac{m-i}{m}$$

The Birthday Paradox

No matter how good our hash function is, we'd better be prepared for collisions because of the birthday paradox.

The probability of there being no collisions after n insertions into an m-element table is

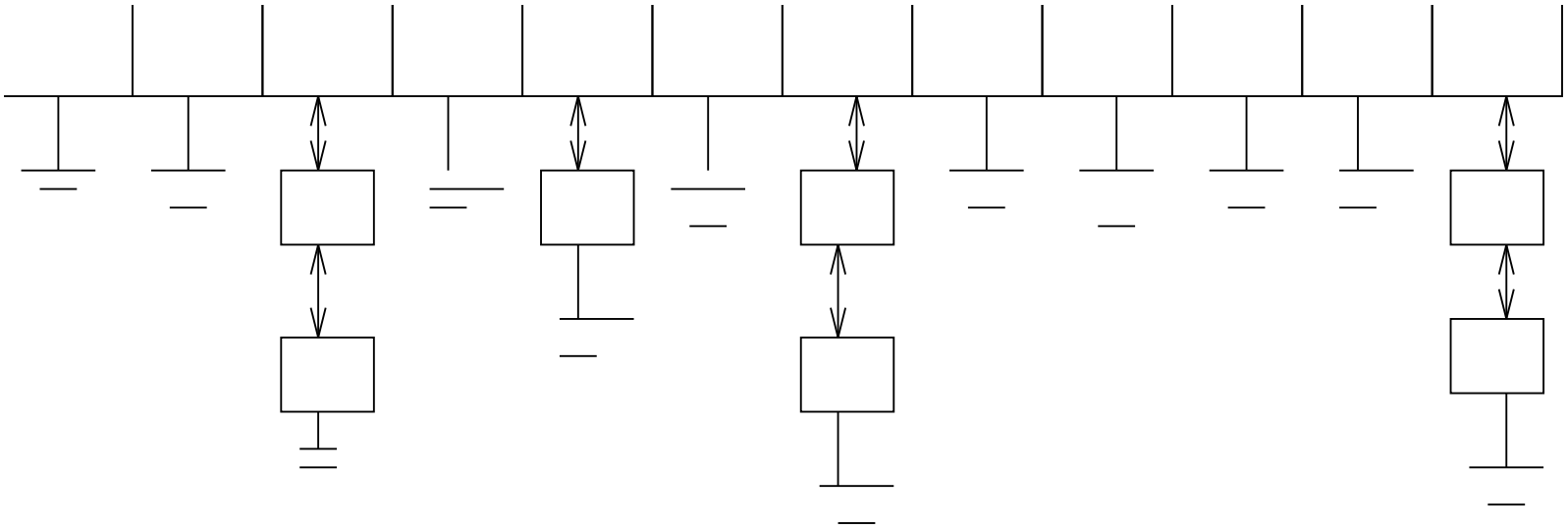
$$\frac{m}{m} \times \frac{m-1}{m} \times \ldots \times \frac{m-n+1}{m} = \prod_{i=0}^{n-1} \frac{m-i}{m}$$

When $m = 365$, this drops below $1/2$ when $n = 23$ and sinks to almost 0 when $n \geq 50$.

Birthday Paradox Example

Collision Resolution by Chaining

The easiest approach is to let each element in the hash table be a pointer to a list of keys.



Chaining

- Insertion, deletion, and search end up becoming linked list ops.

Chaining

- Insertion, deletion, and search end up becoming linked list ops.
- If the n keys are distributed uniformly in a table of size m, each operation takes $O(n/m)$ time.

Chaining

- Insertion, deletion, and search end up becoming linked list ops.
- If the n keys are distributed uniformly in a table of size m, each operation takes $O(n/m)$ time.
- Chaining is easy, but devotes a considerable amount of memory to pointers, which could instead be used to make the table larger.

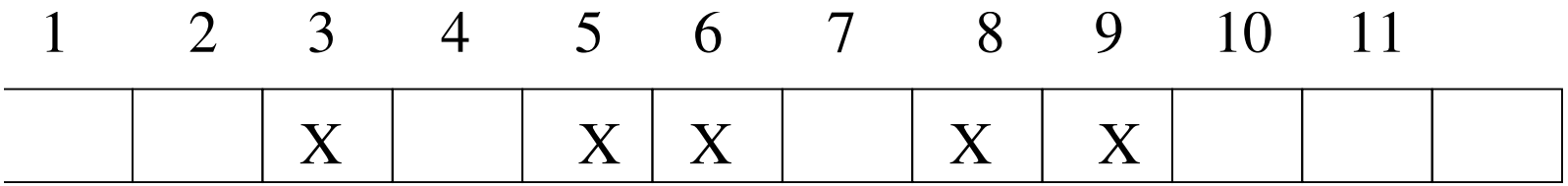
Open Addressing

We can dispense with pointers by using an implicit reference derived from a simple function:

1	2	3	4	5	6	7	8	9	10	11
		X		X	X		X	X		

Open Addressing

We can dispense with pointers by using an implicit reference derived from a simple function:



If the space we want to use is filled, we can examine the remaining locations:

1. Sequentially $h, h+1, h+2, \ldots$
2. Linearly $h, h+k, h+2k, h+3k, \ldots$
3. Quadratically $h, h+1^2, h+2^2, h+3^2 \ldots$

Open Addressing

- The reason for using a more complicated scheme is to avoid long runs from similarly hashed keys.
- Deletion in an open addressing scheme is ugly.

Performance on Dictionary Operations

With either chaining or open addressing:

- Search - $O(1)$ expected, $O(n)$ worst case
- Insert - $O(1)$ expected, $O(n)$ worst case
- Delete - $O(1)$ expected, $O(n)$ worst case
- Min, Max and Predecessor, Successor $\Theta(n+m)$ expected and worst case

Performance on Dictionary Operations

With either chaining or open addressing:

- Search - $O(1)$ expected, $O(n)$ worst case
- Insert - $O(1)$ expected, $O(n)$ worst case
- Delete - $O(1)$ expected, $O(n)$ worst case
- Min, Max and Predecessor, Successor $\Theta(n+m)$ expected and worst case

Pragmatically, a hash table is often the best data structure to maintain a dictionary.

Performance on Dictionary Operations

With either chaining or open addressing:

- Search - $O(1)$ expected, $O(n)$ worst case
- Insert - $O(1)$ expected, $O(n)$ worst case
- Delete - $O(1)$ expected, $O(n)$ worst case
- Min, Max and Predecessor, Successor
  $\Theta(n+m)$ expected and worst case

Pragmatically, a hash table is often the best data structure to maintain a dictionary.
However, we will not use it to analyze algorithms since the worst-case time is unpredictable.

Hashing, Hashing, and Hashing

Udi Manber (CTO) says that the three most important algorithms at Yahoo are

Hashing, Hashing, and Hashing

Udi Manber (CTO) says that the three most important algorithms at Yahoo are hashing,

Hashing, Hashing, and Hashing

Udi Manber (CTO) says that the three most important algorithms at Yahoo are hashing, hashing, and

Hashing, Hashing, and Hashing

Udi Manber (CTO) says that the three most important algorithms at Yahoo are hashing, hashing, and hashing!

Hashing, Hashing, and Hashing

Udi Manber (CTO) says that the three most important algorithms at Yahoo are hashing, hashing, and hashing!
Why?

Hashing, Hashing, and Hashing

Udi Manber (CTO) says that the three most important algorithms at Yahoo are hashing, hashing, and hashing!
Why?
Hashing gives you a short but distinctive representation of a larger document.

Applications

- *Is this new document different from the rest in a large corpus?* – Hash the new document, and compare it to the hash codes of corpus.

Applications

- *Is this new document different from the rest in a large corpus?* – Hash the new document, and compare it to the hash codes of corpus.

Applications

- *Is this new document different from the rest in a large corpus?* – Hash the new document, and compare it to the hash codes of corpus.
- *How can I convince you that a file isn't changed?* – Check if the cryptographic hash code of the file you give me today is the same as that of the original. Note that any changes to the file will result in changing the hash code.

Applications

- *Is this new document different from the rest in a large corpus?* – Hash the new document, and compare it to the hash codes of corpus.
- *How can I convince you that a file isn't changed?* – Check if the cryptographic hash code of the file you give me today is the same as that of the original. Note that any changes to the file will result in changing the hash code.
- Randomized algorithms.