Introduction to Tables
Priority Queues

CS 311 Data Structures and Algorithms

Lecture Slides

Monday, November 16, 2009

Glenn G. Chappell

Department of Computer Science

University of Alaska Fairbanks

CHAPPELLG@member.ams.org

© 2005–2009 Glenn G. Chappell

Review
Where Are We? — The Big Problem

Our problem for much of the rest of the semester:

- Store: a collection of data items, all of the same type.
- Operations:
  - Access items [one item: retrieve/find, all items: traverse].
  - Add new item [insert].
  - Eliminate existing item [delete].
- All this needs to be efficient in both time and space.

A solution to this problem is a **container**.

Generic containers: those in which client code can specify the type of data stored.

Unit Overview
The Basics of Trees

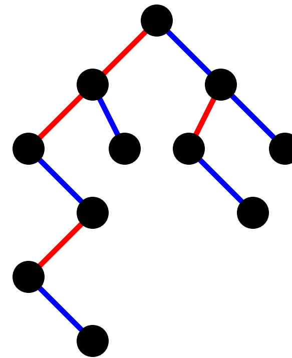
Major Topics

- Introduction to Trees
- Binary Trees
- Binary Search Trees
- Treesort

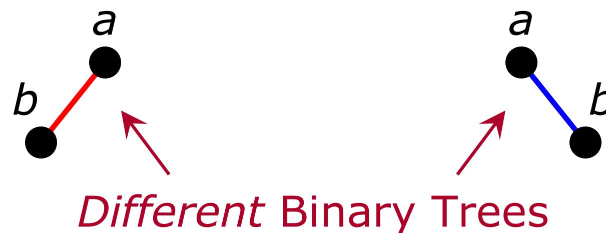
Review
Binary Trees — What a Binary Tree Is

A **Binary Tree** consists of a set *T* of nodes so that either:

- *T* is empty (no nodes), or
- *T* consists of a node *r*, the root, and two subtrees of *r*, each of which is a Binary Tree:
  - the **left subtree**, and
  - the **right subtree**.

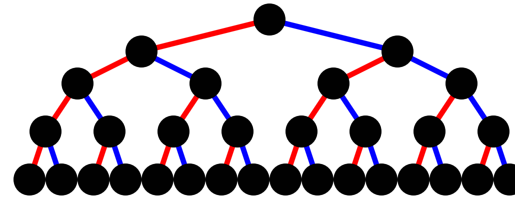


We make a strong distinction between left and right subtrees. Sometimes, we use them for very different things.

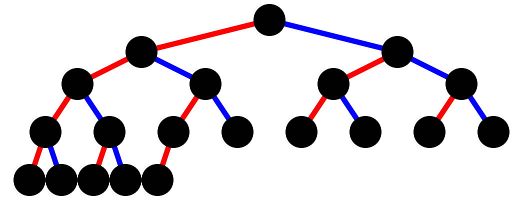


Review
Binary Trees — Three Special Kinds

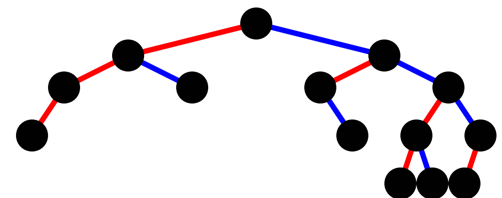
A typical **full** Binary Tree:



A typical **complete** Binary Tree:



A typical **balanced** Binary Tree:



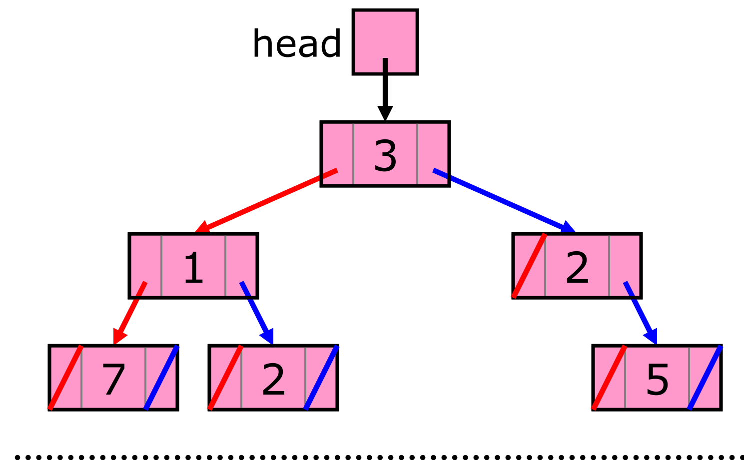
A full Binary Tree is complete; a complete Binary Tree is balanced.

Full → Complete → Balanced

Review
Binary Trees — Implementation

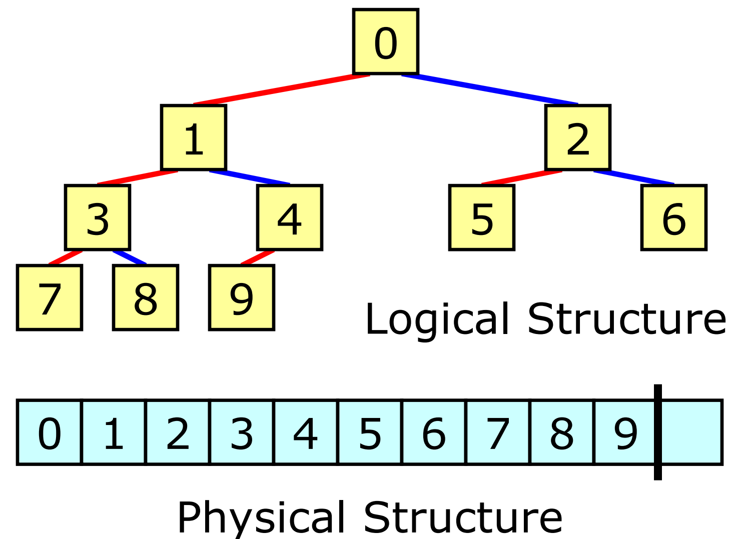
A common way to implement a Binary Tree is to use separately allocated nodes referred to by pointers.

- This is very similar to the way we implemented a Linked List.
- Each node has a data item and two child pointers: left & right.
- A pointer is null if there is no child.



A **complete** Binary Tree can be implemented simply by putting the items in an array, and keeping track of the size of the tree.

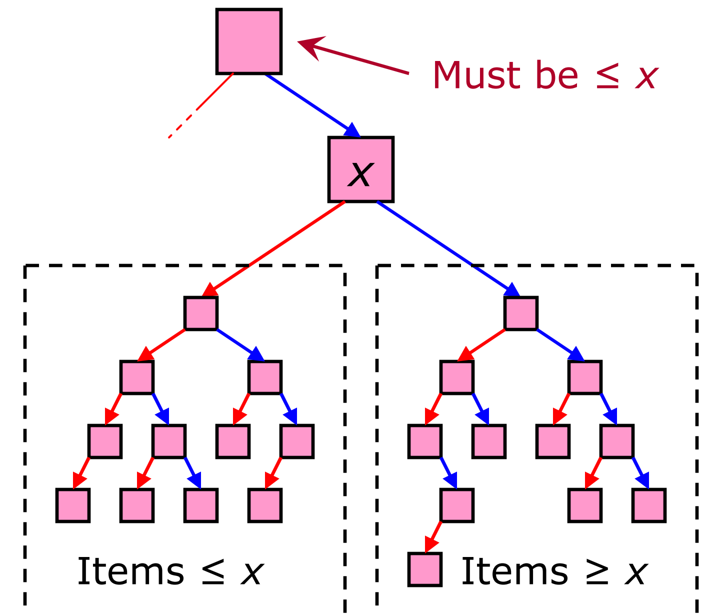
- This is a nice implementation, but it is only useful in situations in which the tree *stays complete*.



Binary Search Trees — What a Binary Search Tree Is

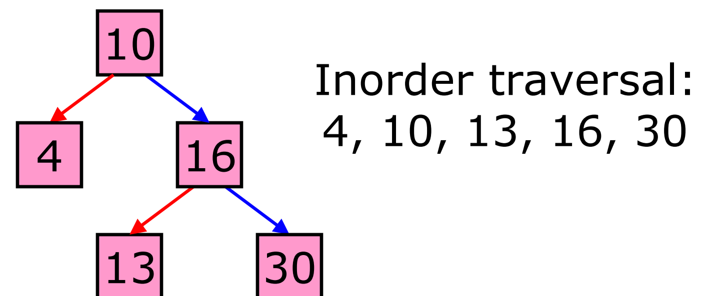
A **Binary Search Tree** is a Binary Tree in which, for each node:

- Descendants holding data less than the node's data are in its left subtree.
- Descendants holding data greater than the node's data are in its right subtree.



In other words, an inorder traversal gives items in sorted order.

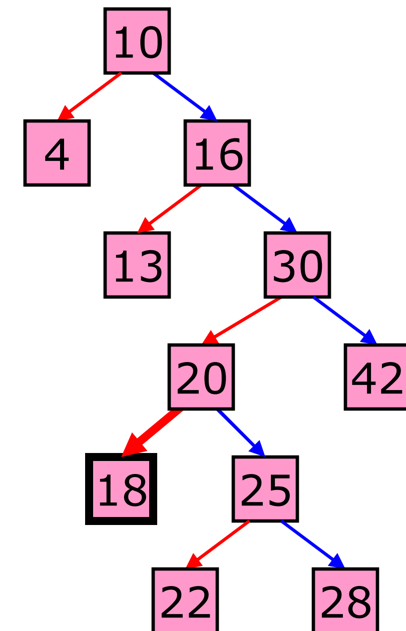
This is another value-oriented ADT (while ADT Binary Tree is position-oriented).



Binary Search Trees
Operations — Retrieve, Insert

To **retrieve** an item in a Binary Search Tree, given its key:

- Begin at the root and repeatedly follow left or right child pointers, depending on how the search key compares to the key in each node.



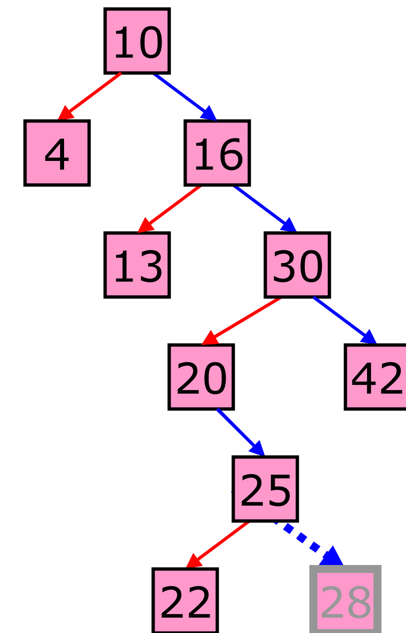
To **insert** a value with a given key:

- Find where the key "should" go (**retrieve** operation).
- Put the value there.

Binary Search Trees
Operations — Delete [1/3]

To **delete** an item from a Binary Search Tree, given its key:

- Find the node (**retrieve** operation).
- Then, three cases:
  - The node to be deleted has no children (it is a leaf).
  - The node has 1 child.
  - The node has 2 children.



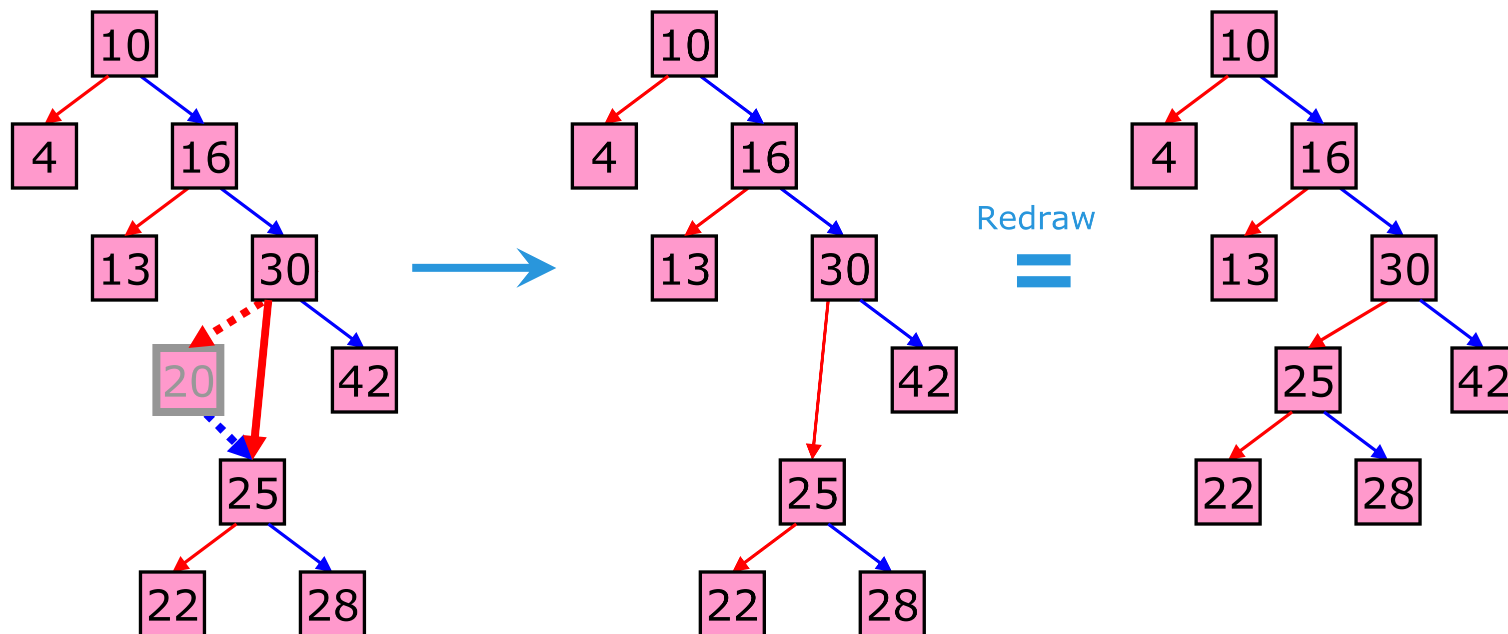
No-child (leaf) case:

- Delete the node, using the appropriate **Binary Tree** operation.

Binary Search Trees
Operations — Delete [2/3]

One-child case:

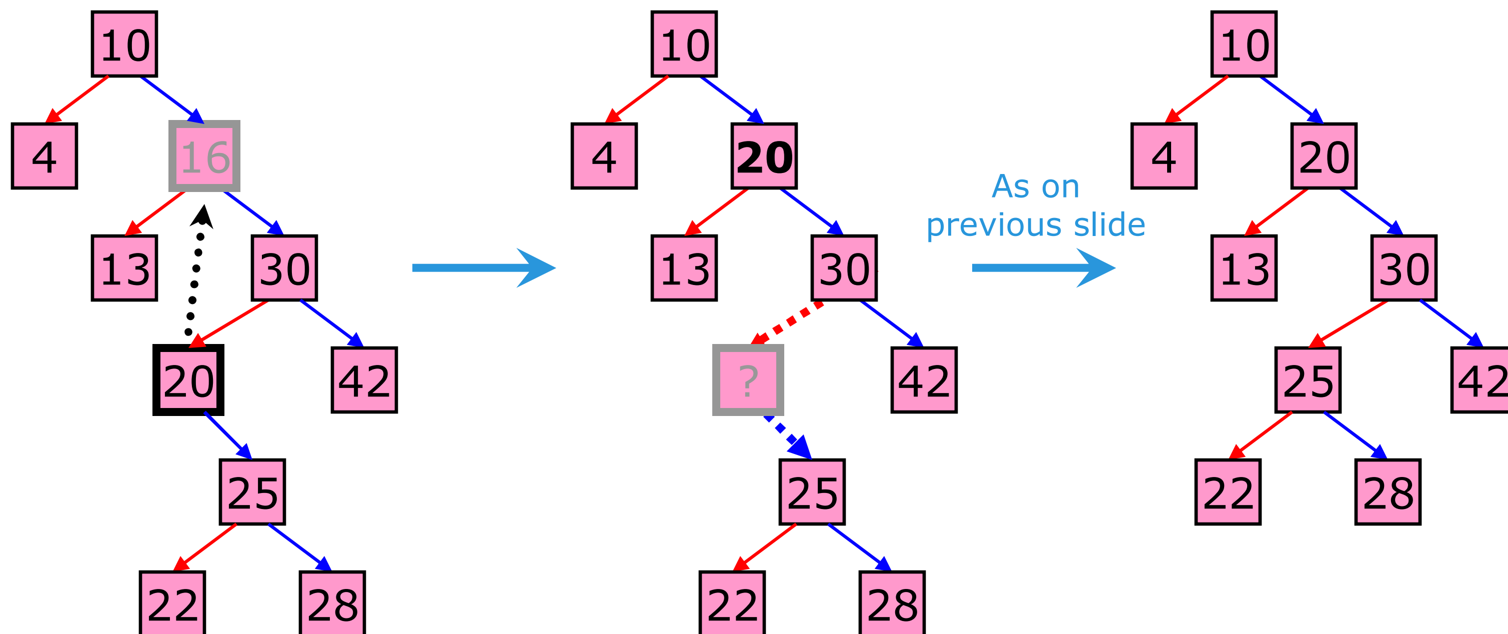
- Replace the subtree rooted at the node with the subtree rooted at its child.
  - This is a constant-time operation, once the node is found.



Binary Search Trees
Operations — Delete [3/3]

Two-child case:

- Replace the data in the node with the data in its **inorder successor** (copy or swap).
- Delete the inorder successor.
    - This has at most one child.



Binary Search Trees
Operations — Summary & Thoughts

Algorithms for the three primary B.S.T. operations:

- Retrieve
  - Start at root. Go down, left or right as appropriate, until either the given key or an empty spot is found.
- Insert
  - Retrieve, then …
  - Put the value in the spot where it should be.
- Delete
  - Retrieve, then …
  - Check number of children node has:
    - 0. Delete node.
    - 1. Replace subtree rooted at node with subtree rooted at child.
    - 2. Copy data from (or swap data with) inorder successor. Proceed as above.

All three operations, in the worst case, require a number of steps proportional to the **height of the tree**.

The height of a tree is small (and all three operations are fast) if the tree is **balanced**.

Review
Binary Search Trees — Efficiency

	B.S.T. (balanced & average case)	Sorted Array	B.S.T. (worst case)
Retrieve	Logarithmic	Logarithmic	Linear
Insert	Logarithmic	Linear	Linear
Delete	Logarithmic	Linear	Linear

Binary Search Trees have poor worst-case performance.

But they have very good performance:

- On average.
- If balanced.
  - But we do not know an efficient way to make them *stay* balanced.

Can we efficiently keep a Binary Search Tree balanced?

- We will look at this question again later.

Unit Overview
Tables & Priority Queues

Major Topics

- Introduction to Tables
- Priority Queues
- Binary Heap algorithms
- Heaps & Priority Queues in the C++ STL
- 2-3 Trees
- Other balanced search trees
- Hash Tables
- Prefix Trees
- Tables in various languages

Introduction to Tables
Types of ADTs

Position-Oriented ADTs

- Get an item based on where it is stored.
- Organize data according to where the client wants it.

Examples

- Sequence
- Stack
- Queue
- Binary Tree

Value-Oriented ADTs

- Get an item based on its value.
  - Or part of the value: key-based look-up.
- Organize data for greatest efficiency.

Examples

- SortedSequence
- Binary Search Tree

Since the client code does not need to know how the data are organized, but only needs efficiency, maybe we can do better here?

Introduction to Tables
Databases

A value-oriented ADT can be thought of an as interface to a general database.

Consider the four data-manipulation commands in the Structured Query Language (SQL):

- **Select**
  - Retrieve a record. Key-based look-up.
- **Update**
  - Change a record.
  - A redundant operation, since we can always delete and then insert. Alternatively, have Select return a reference (or iterator or whatever).
- **Insert**
  - Given a record, insert it.
- **Delete**
  - Given a key, delete the associated record.

These are essentially the operations in a value-oriented ADT.

- We want an implementation that makes them efficient.

Introduction to Tables
Operations — Possibilities

A general value-oriented ADT is called “Table”.

What operations should Table have?

- The Usual
  - **create**, **destroy**, **copy**.
  - **isEmpty**.
  - **size** (maybe).
- Data Manipulation
  - **retrieve** (like SQL Select).
  - *Maybe* **set** (like SQL Update).
    - We generally handle this either by having retrieve return a value in modifiable form, or by simply using delete, then insert.
  - **insert**.
  - **delete**.
- Access All Data
  - **traverse**.

Introduction to Tables
Operations — Issues

Allow multiple items with **equivalent keys**?

- It depends on the requirements of the client.

Require **traverse** to list items in sorted order?

- There are sorted & unsorted implementations. Requiring a sorted traverse would make the latter inefficient.

Allow modification of data while it is in the Table?

- If we have key-data pairs, then modifying the **data** part is no problem.
- Modifying the **key** is tricky, since the item is generally located according to its key. Changing the key means we have to move the item.

Have a separate interface in which the key is the entire value?

- Maybe. Call it “Set”.

Conclusion

- There is no single, best interface to a Table. But they are all very similar.
- Therefore, we will be a little vague about *exactly* what a Table is.

What do we use a Table for?

- To hold data accessed by key fields. For example:
  - Customers accessed by phone number.
  - Students accessed by student ID number.
  - Any other kind of data with an ID code.
- To hold “Set” data.
  - Data in which the only question we ask is whether a key lies in the data set.
- To hold “arrays” whose indices are not nonnegative integers.
  - `arr2["hello"] = 3;`
- To hold array-like data sets that are **sparse**.
  - `arr[6] = 1; arr[1000000000] = 2;`

Introduction to Tables
Possible Implementations [1/3]

What are possible Table implementations?

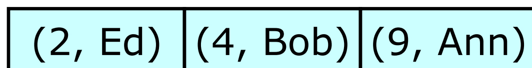
- A Sequence holding key-data pairs.
  - Sorted or unsorted.
  - Array-based or Linked-List-based.
- A Binary Search Tree holding key-data pairs.
  - Implemented using a pointer-based Binary Tree.

Table

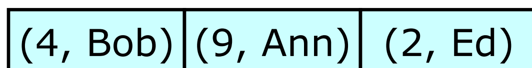
Key	Data
4	Bob
9	Ann
2	Ed

Array Implementations

Sorted

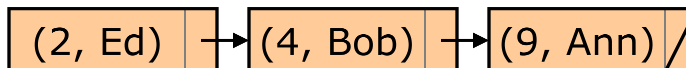


Unsorted

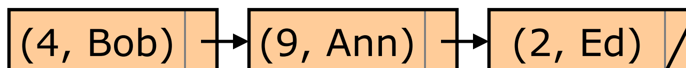


Linked List Implementations

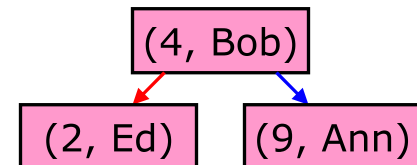
Sorted



Unsorted



Binary Search Tree Implementation



Introduction to Tables
Possible Implementations [2/3]

Find the order of each operation, for the following Table implementations.

- Allow multiple equivalent keys, where it matters. (Otherwise, when inserting, we must always do a retrieve operation first.)

	Sorted Array	Unsorted Array	Sorted Linked List	Unsorted Linked List	Binary Search Tree	Balanced*** Binary Search Tree
Retrieve	Logarithmic	Linear	Linear	Linear	Linear	Logarithmic
Insert	Linear	Amortized constant (or constant)*	Linear**	Constant	Linear	Logarithmic
Delete	Linear	Linear	Linear**	Linear**	Linear	Logarithmic

*Constant time if we have pre-allocated enough storage.

**We must find the location first (retrieve operation); that requires linear time.

***We do not—yet—know any way to guarantee that the tree will *stay* balanced, unless we can restrict ourselves to read-only operations (no insert, delete).

Introduction to Tables
Possible Implementations [3/3]

Tables can be implemented in many ways.

- Different implementations are appropriate in different circumstances.

In special situations, the (amortized) constant-time insertion for an unsorted array and the logarithmic-time retrieve for a sorted array can be combined!

- Insert all data into an unsorted array, sort the array, then use Binary Search to retrieve data.
- This is a good way to handle Table data with **separate filling & searching phases** (and few or no deletes).
- Note: We will be talking about some complicated Table implementations. But *sometimes* a simple solution is the best.

Introduction to Tables
Better Ideas? [1/3]

	Sorted Array	Unsorted Array	Sorted Linked List	Unsorted Linked List	Binary Search Tree	Balanced (how?) Binary Search Tree
Retrieve	Logarithmic	Linear	Linear	Linear	Linear	Logarithmic
Insert	Linear	Constant???	Linear	Constant	Linear	Logarithmic
Delete	Linear	Linear	Linear	Linear	Linear	Logarithmic

Idea #1: Restricted Table

- Perhaps we can do better if we do not implement a Table in its full generality.
- Maybe do not allow retrieve & delete on *all* keys.

In practice: Priority Queue

Introduction to Tables
Better Ideas? [2/3]

	Sorted Array	Unsorted Array	Sorted Linked List	Unsorted Linked List	Binary Search Tree	Balanced (how?) Binary Search Tree
Retrieve	Logarithmic	Linear	Linear	Linear	Linear	Logarithmic
Insert	Linear	Constant???	Linear	Constant	Linear	Logarithmic
Delete	Linear	Linear	Linear	Linear	Linear	Logarithmic

Idea #2: Keep a Tree Balanced

- Figure out how to keep a Binary Search Tree balanced, without compromising efficiency.
- Maybe loosen the “binary” requirement.
- Maybe loosen the “balanced” requirement, too.

In practice: Balanced search trees

- 2-3 Tree & 2-3-4 Tree
- **Red-Black Tree**
- AVL Tree
- **B-Tree & B+-Tree**

Introduction to Tables
Better Ideas? [3/3]

	Sorted Array	Unsorted Array	Sorted Linked List	Unsorted Linked List	Binary Search Tree	Balanced (how?) Binary Search Tree
Retrieve	Logarithmic	Linear	Linear	Linear	Linear	Logarithmic
Insert	Linear	Constant???	Linear	Constant	Linear	Logarithmic
Delete	Linear	Linear	Linear	Linear	Linear	Logarithmic

Idea #3: "Magic Functions"

- Consider the simplest structure possible: unsorted array.
  - Arrays have fast look-up by index. So …
  - … a "magic function" that gives us an key's index might make things very fast.
  - Ah, but what about deleting? Idea: Allow **empty** items, so that we do not need to move things down when we delete.
  - It looks like we might be able to insert, delete, and retrieve in **constant time**.
- But can we find a magic function?

In practice: Hash Tables

Priority Queues
What a Priority Queue Is — Introduction

Our next ADT is **Priority Queue**.

- It has almost the same operations as Queue.
- The difference is that each data item has a key, called its **priority**. The item retrieved/deleted is the item with the highest priority.

Conceptually, a Priority Queue is another way to do “standing in line”.

- However, items are not handled in the order they were inserted, but rather in order of priority.

Thus, a Priority Queue is not a Queue!

Priority Queues
What a Priority Queue Is — Restricted Table

We have discussed turning a Sequence into a Queue.

- In a Sequence, we retrieve/delete at **any given position**.
- In a Queue, we can retrieve/delete only the element at the **highest position**.

We can similarly turn a (sorted) Table into a Priority Queue.

- In a Table, we retrieve/delete the item with **any given key**.
- In a Priority Queue, we can retrieve/delete only the element with the **highest key** (priority).

Thus, a Priority Queue is a restricted-access (sorted) Table, just as a Queue is a restricted-access Sequence.

Priority Queues
What a Priority Queue Is — ADT

Priority Queue has the following data and operations. (These differ a little from those in the text.)

- Data
  - A collection of items, each of which has a priority.
- Operations
  - The Usual
    - **create**, **destroy**, **copy**.
    - **isEmpty**.
  - Operations Specific to Priority Queues
    - **insert**. Given an item (which includes a priority).
    - **getFront**. Gets item with highest priority.
    - **delete**. Removes item with highest priority.

Priority Queues
Applications

A PQ is useful when we have items to process and some are more important than others.

By cleverly choosing priorities, we can produce "hybrid" structures.

- If an item's priority includes a time stamp, then we can simulate a mixed Queue/PQ.
  - Items with the same priority can be dealt with in FIFO order.
- Or a mixed Stack/PQ, if you like.
  - What is this good for? I couldn't say.

PQs can be used to do sorting.

- Insert all items, then retrieve/delete all items. The resulting sequence is sorted by priority.
- We can also use a PQ to handle special cases, in which the data to be sorted are modified during sorting.
- Note: Once again, a sorted container gives us a sorting algorithm. However, as with Insertion Sort, instead of using a separate container to sort with, we prefer to use an in-place version of the algorithm. We will call this "Heap Sort".

Priority Queues
Implementation

We can implement a Priority Queue using any of the methods we have discussed for implementing a Table.

- And they are all still just as dissatisfying. ☹
- In particular, they all have linear-time delete.

The most interesting thing about PQs is the way they are *usually* implemented, using a structure called a (Binary) Heap.

- We discuss this next …