Using a heap as a priority queue

- To remove highest priority item from heap:
  `remove root;` / * `O(1)` complexity */
  Move last item to root, then …
  `heapify in reverse;` / * `O(log n)` complexity */
  - So overall complexity is `O(log n)`
- Also `O(log n)` for insert function
- Compare to other priority queue strategies
  - Sorted list: insert – `O(n)`; remove – `O(1)`
  - Unsorted array: insert – `O(1)`; remove – `O(n)`
- Choose heap strategy if `n` is expected to be large

A table ADT

- Declare Table type (define in implementation)
  `typedef struct TableTag Table;`
  - Also define a `KeyType`, and maybe a `DataType` (or just use `void *`)
- Let user define initial size of table
  `Table *createTable(int startingSize);`
- Can put/get/update/remove info associated with unique key

```
int put(KeyType key, void *info, Table *table);
void* get(KeyType key, Table *table);
int update(KeyType key, void *info, Table *table);
int remove(KeyType key, Table *table);
```

  - Functions return false if unsuccessful (except `get` returns `NULL`)
- Can print all info, usually in key order
  `void printAll(Table *table);`

Table implementation options

- Many possibilities – depends on application
  - And how much trouble efficiency is worth
- Option 1: use a BST
  - To put: insertTree using key for ordering
  - To update: deleteTree, then insertTree
  - To getAll: use in-order traversal
- Option 2: sorted array with binary searching (later)
- Option 3: implement as a "hash table"
  - Hashing – general technique works great with tables

Hashing ideas and concepts

- Idea: transform arbitrary key domain (e.g., strings) into "dense integer range" – then use result as index to table
  - `index = hash(key);` /* function returns int */
- Collisions: `hash(k1)==hash(k2), k1 != k2`
  - Usually impossible to avoid ("perfect hashing"), so must have a way to handle collisions
  - e.g., probe for empty slot if using "open addressing" - `while (!empty(index)) index = probe(key);`
- Concept: insertion/searching is quick – but only until the table starts to get filled up
  - Then collisions start happening too often!

Open address hashing
– & implementing basic table ADT

- Define structs for table items and whole table of items

```
typedef struct
    { KeyType key; void *info; } TableItem;
typedef struct
    { int size; int n; TableItem *items; } Table;
```

  - `size` is size of array; `n` is the number of items in the table
  - Constructor allocates memory for array of items, and initializes all items to "empty" key
- The `put` function uses `hash(key)` and `probe(key)` to find empty slot for new item
  - *Resizes* array (and *rehashes* existing items) whenever table "load factor" reaches 50 percent (rule of thumb for open addressing)

Open address hashing (cont.)

- `get` & `update` functions use `hash(key)` and `probe(key)` in *exact same sequence* as `put` – to find existing info
- `remove` is more complicated
  - Cannot just remove an item – future probes for get and update might terminate prematurely at empty slot
  - Inefficient technique rehashes all items
  - Alternative technique uses "deleted" key markers
    - But problem with that is table fills up prematurely
- `printAll` in key order – must sort first!
  - So `O(n log n)` at best!

Resolving collisions

- Simplest open address approach is linear probing
  - If (`index = hash(key)`) is not empty, try `index+1`, then `index+2`, …, until empty slot
  - In other words, searching for first “open address”
  - Biggest problem: it leads to “primary clusters”
- Quadratic probing – varies probe, like 1, 3, 6, …
  - Leads to “secondary clusters” but not as quickly
- Double hashing – `probe(key)` varies by key
  - Best open addressing approach for avoiding clusters
- Or a completely different approach: “chaining”

Chaining

- Table is an array of pointers to lists:

```
typedef struct TableTag
        { int size; int n;
          ListPointer *lists; };
```

- Constructor allocates memory for array, and creates an empty list for each element of the array
- `put` function uses `hash(key)` and appends to end of list
  - Clustering not a problem, but long lists can be, so rule of thumb is resize when load factor approaches 80%
- `remove` function is easier now – just delete from list
- But lots more overhead than open addressing
  - Must store node pointers as well as key and info
  - Use list function calls instead of direct array access

Recursive binary searching

- Start with a sorted array: `a[0..n-1]`
  - Useful item in a is `struct{key, info} ItemType`;
- Binary searching algorithm is naturally recursive:

```
int bsearch(KeyType key, ItemType a[],
            int left, int right) {
   /* first call is for left=0, and right=n-1 */
   int middle = (left + right) / 2;
   if (key == a[middle].key) return middle; /* success */
   if (left > right) return -1; /* unsuccessful */
   if (key > a[middle].key) /* search one half or the other */
      return bsearch(key, a, middle+1, right);
   else return bsearch(key, a, left, middle-1);
}
```

- Iterative version is a little trickier (but not too hard)

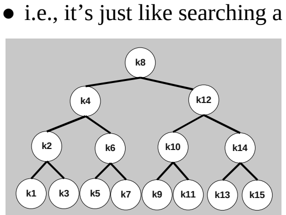
Iterative binary searching

```
int bsearch(KeyType key, ItemType a[], int n) {
   int low = 0, high = n-1, middle;
   while (low <= high) {
      middle = (low + high) / 2;
      if (key == a[middle].key)
         return middle; /* success */
      if (key > a[middle].key) low = middle + 1;
      else high = middle – 1;
   }
   return -1; /* unsuccessful */
}
```

- Both versions are same complexity class
  - But recursive version has more overhead, so actually runs a bit slower than iterative version
  - Interpolation search, by the way, is in a faster class

Complexity of binary search

- Say array has 15 elements, $k_1..k_{15}$: `a[0..14]`
  - If key is at k_8 (`a[7]`) then found by 1 comparison
  - If key is at k_4 or k_{12}, takes 3 comparisons …
- i.e., it’s just like searching a BST



- Problem size is halved at each step
  - So complexity class is `O(log n)`
- Interpolation search reduces more quickly
  - Class is `O(log log n)`

Compare 3 table implementations

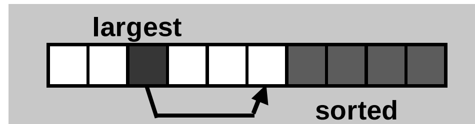
Table operation	Hash table	BST	Sorted array
create	`O(n)`	`O(1)`	`O(n)`
find, get, update	`O(1)`	`O(log n)`	`O(log n)`
put	`O(1)`	`O(log n)`	`O(n)`
remove	`O(1)`	`O(log n)`	`O(n)`
printAll	`O(n log n)`	`O(n)`	`O(n)`

- Conclusion – depends on table purpose & `n` size
  - Hash table wins for most applications if n is large
  - BST wins if expect to printAll frequently
  - Sorted array might win for small n – to minimize overhead/work

Sorting

- Probably *the* most expensive common operation
- Problem: arrange `a[0..n-1]` by some ordering
  - e.g., in ascending order: `a[i-1]<=a[i], 0<i<n`
- Two general types of strategies
  - Comparison-based sorting – includes most strategies
    - Apply to any comparable data – (key, info) pairs
    - Lots of simple, inefficient algorithms
    - Some not-so-simple, but more efficient algorithms
  - Address calculation sorting – rarely used in practice
    - Must be tailored to fit the data – not all data are suitable
    - Won't cover in CS 12 – see proxmap and radix sorts in sec. 13.6

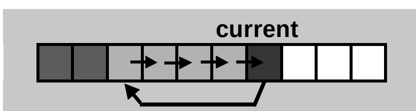
Selection sort



- Idea: build sorted sequence at end of array
- At each step:
  - *Find largest* value in not-yet-sorted portion
  - *Exchange* this value with the one at end of unsorted portion (now beginning of sorted portion)
- Complexity is $O(n^2)$ – but simple to program
  - Also – best way to find k^{th} largest, or top k values

Insertion sort

- Generally "better" than other simple algorithms
- Inserts one element into sorted part of array
  - Must move other elements to make room for it



- Complexity is $O(n^2)$ (code)
  - But runs faster than selection sort and others in class
  - Really quick on *nearly sorted* array
- Often used to supplement more sophisticated sorts

Divide & conquer strategies

- Idea: (1) divide array in two; (2) sort each part; (3) combine two parts to overall solution
- e.g., mergeSort

```
if (more than one item in array):
   divide array into left half and right half;
   mergeSort(left half); mergeSort(right half);
   merge(left half and right half together);
```

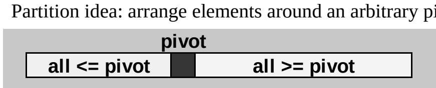
  - Requires helper method to merge two halves
  - Complexity is $O(n \log n)$
  - The best sort for large files (too big for memory)
- But for most problems, quickSort is a better divide & conquer strategy

Quick sort

- Basic quicksort algorithm is recursive

```
if (there is something to sort)
{    partition array elements;
     sort left part; sort right part;
} /* It's the utility partition function that does all the work! */
```

- Partition idea: arrange elements around an arbitrary pivot




```
scan from (i = left) until a[i] >= pivot;
scan from (j = right) until a[j] <= pivot;
swap a[i], a[j];
continue both scans until i > j;
```

(code)

Quick sort (cont.)

- Complexity is $O(n \log n)$ on average
  - Fastest comparison-based sorting algorithm
  - But overkill, and not-so-fast with small arrays
    - One frequently-used optimization applies insertion sort for partitions smaller than than 10 or so
- But worst case is $O(n^2)$!
  - Just like BST worst case – sorted order can be bad
    - Especially if first or last is chosen as pivot – middle is better
- By the way – see `qsort` in `<stdlib.h>` (code)
  - Also by the way – see $O(n)$ address calculation sorts if really fast sorting is required for an application