6.087 Lecture 9 – January 22, 2010

- Review
- Using External Libraries
  - Symbols and Linkage
  - Static vs. Dynamic Linkage
  - Linking External Libraries
  - Symbol Resolution Issues
- Creating Libraries
- Data Structures
  - B-trees
  - Priority Queues

Review: Void pointers

- Void pointer – points to any data type:

```
int x; void * px = &x; /* implicit cast to (void *) */
float f; void * pf = &f;
```

- Cannot be dereferenced directly; void pointers must be cast prior to dereferencing:

```
printf("%d %f\n", *(int *)px, *(float *)pf);
```

Review: Function pointers

- Functions not variables, but also reside in memory (i.e. have an address) – we can take a pointer to a function
- Function pointer declaration:

```
int (*cmp)(void *, void *);
```

- Can be treated like any other pointer
- No need to use & operator (but you can)
- Similarly, no need to use * operator (but you can)

Review: Function pointers

```
int strcmp_wrapper(void * pa, void * pb) {
  return strcmp((const char *)pa, (const char *)pb);
}
```

- Can assign to a function pointer:

  `int (*fp)(void *, void *) = strcmp_wrapper;` or

  `int (*fp)(void *, void *) = &strcmp_wrapper;`

- Can call from function pointer: (`str1` and `str2` are strings)

  `int ret = fp(str1, str2);` or

  `int ret = (*fp)(str1, str2);`

Review: Hash tables

- Hash table (or hash map): array of linked lists for storing and accessing data efficiently
- Each element associated with a key (can be an integer, string, or other type)
- Hash function computes hash value from key (and table size); hash value represents index into array
- Multiple elements can have same hash value – results in collision; elements are chained in linked list

6.087 Lecture 9 – January 22, 2010

- Review
- Using External Libraries
  - Symbols and Linkage
  - Static vs. Dynamic Linkage
  - Linking External Libraries
  - Symbol Resolution Issues
- Creating Libraries
- Data Structures
  - B-trees
  - Priority Queues

Symbols and libraries

- External libraries provide a wealth of functionality – example: C standard library
- Programs access libraries' functions and variables via identifiers known as *symbols*
- Header file declarations/prototypes mapped to symbols at compile time
- Symbols linked to definitions in external libraries during *linking*
- Our own program produces symbols, too

Functions and variables as symbols

- Consider the simple hello world program written below:

```
#include <stdio.h>

const char msg[] = "Hello, world.";

int main(void) {
  puts(msg);
  return 0;
}
```

- What variables and functions are declared globally?

Functions and variables as symbols

- Consider the simple hello world program written below:

```
#include <stdio.h>

const char msg[] = "Hello, world.";

int main(void) {
  puts(msg);
  return 0;
}
```

- What variables and functions are declared globally?

  `msg`, `main()`, `puts()`, others in `stdio.h`

Functions and variables as symbols

- Let's compile, but not link, the file hello.c to create hello.o:

```
athena%[1] gcc -Wall -c hello.c -o hello.o
```

  - `-c`: compile, but do not link hello.c; result will compile the code into machine instructions but not make the program executable
  - addresses for lines of code and static and global variables not yet assigned
  - need to perform *link* step on `hello.o` (using `gcc` or `ld`) to assign memory to each symbol
  - linking resolves symbols defined elsewhere (like the C standard library) and makes the code executable

[1] Athena is MIT's UNIX-based computing environment. OCW does not provide access to it.

Functions and variables as symbols

- Let's look at the symbols in the compiled file hello.o:

  `athena%`[1] `nm hello.o`
- Output:

```
0000000000000000 T main
0000000000000000 R msg
                 U puts
```

- 'T' – (text) code; 'R' – read-only memory; 'U' - undefined symbol
- Addresses all zero before linking; symbols not allocated memory yet
- Undefined symbols are defined externally, resolved during linking

[1] Athena is MIT's UNIX-based computing environment. OCW does not provide access to it.

Functions and variables as symbols

- Why aren't symbols listed for other declarations in `stdio.h`?
- Compiler doesn't bother creating symbols for unused function prototypes (saves space)
- What happens when we link?

  `athena%`[1] `gcc -Wall hello.o -o hello`

  - Memory allocated for defined symbols
  - Undefined symbols located in external libraries (like `libc` for C standard library)

[1] Athena is MIT's UNIX-based computing environment. OCW does not provide access to it.

Functions and variables as symbols

- Let's look at the symbols now:

```
athena%1 nm hello
```

- Output:
  (other default symbols)
  ⋮

```
0000000000400524 T main
000000000040062c R msg
                 U puts@@GLIBC_2.2.5
```

- Addresses for static (allocated at compile time) symbols
- Symbol `puts` located in shared library GLIBC_2.2.5 (GNU C standard library)
- Shared symbol `puts` not assigned memory until run time

[1] Athena is MIT's UNIX-based computing environment. OCW does not provide access to it.

Static and dynamic linkage

- Functions, global variables must be allocated memory before use
- Can allocate at compile time (static) or at run time (shared)
- Advantages/disadvantages to both
- Symbols in same file, other `.o` files, or static libraries (archives, `.a` files) – static linkage
- Symbols in shared libraries (`.so` files) – dynamic linkage
- `gcc` links against shared libraries by default, can force static linkage using `-static` flag

Static linkage

- What happens if we statically link against the library?

```
athena%[1] gcc -Wall -static hello.o -o hello
```

- Our executable now contains the symbol `puts`:

```
⋮
00000000004014c0 W puts
⋮
0000000000400304 T main
⋮
000000000046cd04 R msg
⋮
```

- 'W': linked to another defined symbol

[1] Athena is MIT's UNIX-based computing environment. OCW does not provide access to it.

Static linkage

- At link time, statically linked symbols added to executable
- Results in much larger executable file (static – 688K, dynamic – 10K)
- Resulting executable does not depend on locating external library files at run time
- To use newer version of library, have to recompile

Dynamic linkage

- Dynamic linkage occurs at run-time
- During compile, linker just looks for symbol in external shared libraries
- Shared library symbols loaded as part of program startup (before `main()`)
- Requires external library to define symbol exactly as expected from header file declaration
  - changing function in shared library can break your program
  - version information used to minimize this problem
  - reason why common libraries like `libc` rarely modify or remove functions, even broken ones like `gets()`

Linking external libraries

- Programs linked against C standard library by default
- To link against library `lib`*namespec*`.so` or `lib`*namespec*`.a`, use compiler flag `-l`*namespec* to link against library
- Library must be in library path (standard library directories + directories specified using `-L` *directory* compiler flag
- Use `-static` for force static linkage
- This is enough for static linkage; library code will be added to resulting executable

Loading shared libraries

- Shared library located during compile-time linkage, but needs to be located again during run-time loading
- Shared libraries located at run-time using linker library `ld.so`
- Whenever shared libraries on system change, need to run `ldconfig` to update links seen by `ld.so`
- During loading, symbols in dynamic library are allocated memory and loaded from shared library file

Loading shared libraries on demand

- In Linux, can load symbols from shared libraries on demand using functions in `dlfcn.h`
- Open a shared library for loading:

```c
void * dlopen(const char *file, int mode);
```

  values for mode: combination of `RTLD_LAZY` (lazy loading of library), `RTLD_NOW` (load now), `RTLD_GLOBAL` (make symbols in library available to other libraries yet to be loaded), `RTLD_LOCAL` (symbols loaded are accessible only to your code)

Loading shared libraries on demand

- Get the address of a symbol loaded from the library:

  ```
  void * dlsym(void * handle, const char * symbol_name);
  ```

  handle from call to dlopen; returned address is pointer to variable or function identified by `symbol_name`

- Need to close shared library file handle after done with symbols in library:

  ```
  int dlclose(void * handle);
  ```

- These functions are not part of C standard library; need to link against library `libdl`: `-ldl` compiler flag

Symbol resolution issues

- Symbols can be defined in multiple places
- Suppose we define our own `puts()` function
- But, `puts()` defined in C standard library
- When we call `puts()`, which one gets used?

Symbol resolution issues

- Symbols can be defined in multiple places
- Suppose we define our own `puts()` function
- But, `puts()` defined in C standard library
- When we call `puts()`, which one gets used?
- Our `puts()` gets used since ours is static, and `puts()` in C standard library not resolved until run-time
- If statically linked against C standard library, linker finds two `puts()` definitions and aborts (multiple definitions not allowed)

Symbol resolution issues

- How about if we define `puts()` in a shared library and attempt to use it within our programs?
- Symbols resolved in order they are loaded
- Suppose our library containing `puts()` is `libhello.so`, located in a standard library directory (like `/usr/lib`), and we compile our `hello.c` code against this library:

  `athena%`[1] `gcc -g -Wall hello.c -lhello -o hello.o`
- Libraries specified using `-l` flag are loaded in order specified, and before C standard library
- Which `puts()` gets used here?

  `athena% gcc -g -Wall hello.c -lc -lhello -o hello.o`

[1] Athena is MIT's UNIX-based computing environment. OCW does not provide access to it.

6.087 Lecture 9 – January 22, 2010

- Review
- Using External Libraries
  - Symbols and Linkage
  - Static vs. Dynamic Linkage
  - Linking External Libraries
  - Symbol Resolution Issues
- Creating Libraries
- Data Structures
  - B-trees
  - Priority Queues

Creating libraries

- Libraries contain C code like any other program
- Static or shared libraries compiled from (un-linked) object files created using `gcc`
- Compiling a static library:
  - compile, but do not link source files:
    `athena%`[1] `gcc -g -Wall -c` *`infile.c`* `-o` *`outfile.o`*
  - collect compiled (unlinked) files into an archive:
    `athena% ar -rcs lib`*`name`*`.a` *`outfile1.o`* *`outfile2.o`* `...`

[1] Athena is MIT's UNIX-based computing environment. OCW does not provide access to it.

Creating shared libraries

- Compile and do not link files using `gcc`:

  `athena%`[1] `gcc -g -Wall -fPIC -c` *`infile.c`* `-o` *`outfile.o`*
- `-fPIC` option: create position-independent code, since code will be repositioned during loading
- Link files using `ld` to create a shared object (`.so`) file:

  `athena% ld -shared -soname lib`*`name`*`.so -o lib`*`name`*`.so.`*`version`* `-lc` *`outfile1.o outfile2.o`* `...`
- If necessary, add directory to `LD_LIBRARY_PATH` environment variable, so `ld.so` can find file when loading at run-time
- Configure `ld.so` for new (or changed) library:

  `athena% ldconfig -v`

[1] Athena is MIT's UNIX-based computing environment. OCW does not provide access to it.

6.087 Lecture 9 – January 22, 2010

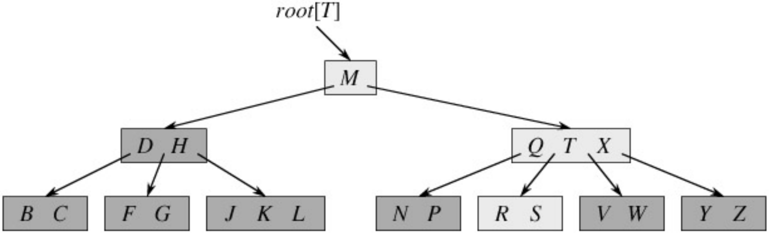
- Review
- Using External Libraries
  - Symbols and Linkage
  - Static vs. Dynamic Linkage
  - Linking External Libraries
  - Symbol Resolution Issues
- Creating Libraries
- Data Structures
  - B-trees
  - Priority Queues

Data structures

- Many data structures designed to support certain algorithms
- B-tree - generalized binary search tree, used for databases and file systems
- Priority queue - ordering data by “priority,” used for sorting, event simulation, and many other algorithms

B-tree structure

- Binary search tree with variable number of children (at least t, up to $2t$)
- Tree is balanced – all leaves at same level
- Node contains list of "keys" – divide range of elements in children



[Cormen, Leiserson, Rivest, and Stein. *Introduction to Algorithms*, 2nd ed. MIT Press, 2001.]

Initializing a B-tree

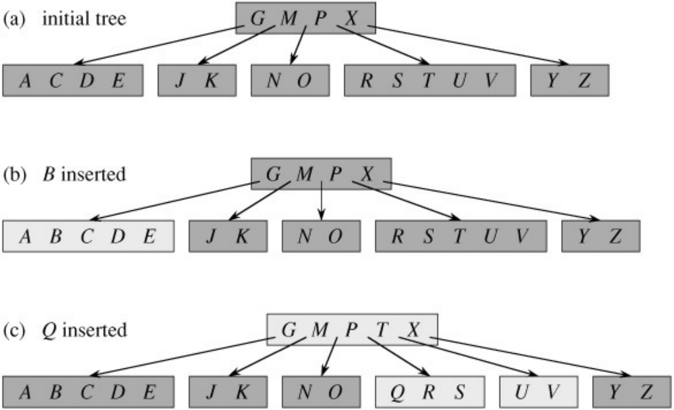
- Initially, B-tree contains root node with no children (leaf node), no keys
- Note: root node exempt from minimum children requirement

Inserting elements

- Insertion complicated due to maximum number of keys
- At high level:
    1. traverse tree down to leaf node
    2. if leaf already full, split into two leaves:
        (a) move median key element into parent (splitting parent already full)
        (b) split remaining keys into two leaves (one with lower, one with higher elements)
    3. add element to sorted list of keys
- Can accomplish in one pass, splitting full parent nodes during traversal in step 1

Inserting elements

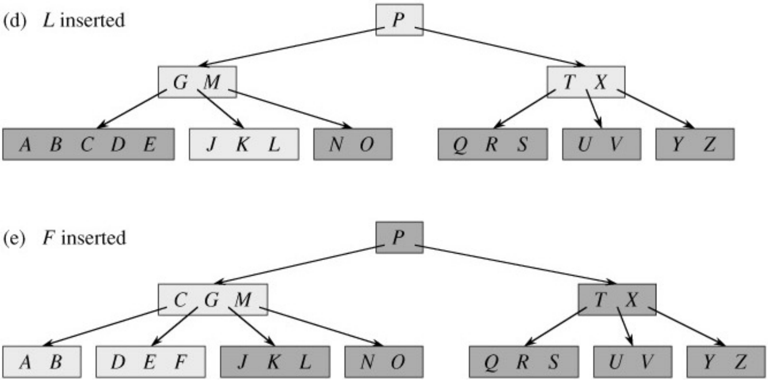
B-tree with $t = 3$ (nodes may have 2–5 keys):



[Cormen, Leiserson, Rivest, and Stein. *Introduction to Algorithms*, 2nd ed. MIT Press, 2001.]

Inserting elements

More insertion examples:



[Cormen, Leiserson, Rivest, and Stein. *Introduction to Algorithms*, 2nd ed. MIT Press, 2001.]

Searching a B-tree

- Search like searching a binary search tree:
  1. start at root.
  2. if node empty, element not in tree
  3. search list of keys for element (using linear or binary search)
  4. if element in list, return element
  5. otherwise, element between keys, and repeat search on child node for that range
- Tree is balanced – search takes O$(\log n)$ time

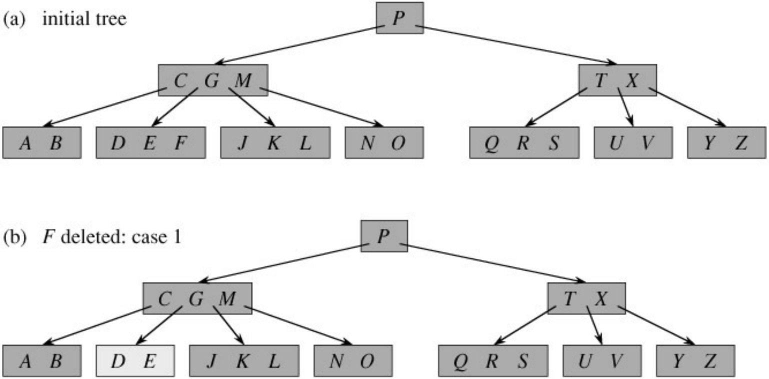
Deletion

- Deletion complicated by minimum children restriction
- When traversing tree to find element, need to ensure child nodes to be traversed have enough keys
  - if adjacent child node has at least t keys, move separating key from parent to child and closest key in adjacent child to parent
  - if no adjacent child nodes have extra keys, merge child node with adjacent child
- When removing a key from a node with children, need to rearrange keys again
  - if child before or after removed key has enough keys, move closest key from child to parent
  - if neither child has enough keys, merge both children
  - if child not a leaf, have to repeat this process

Deletion examples

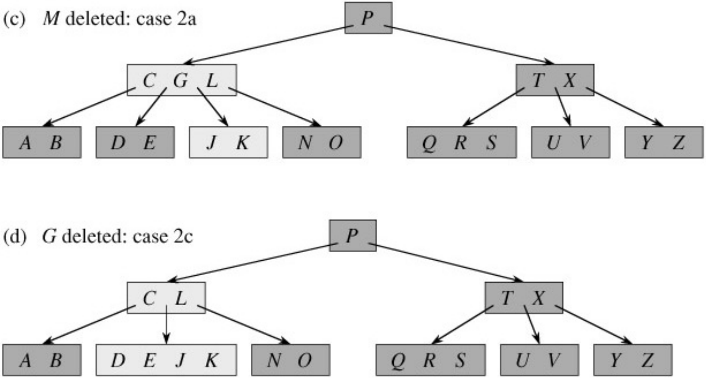
(a) initial tree

(b) F deleted: case 1



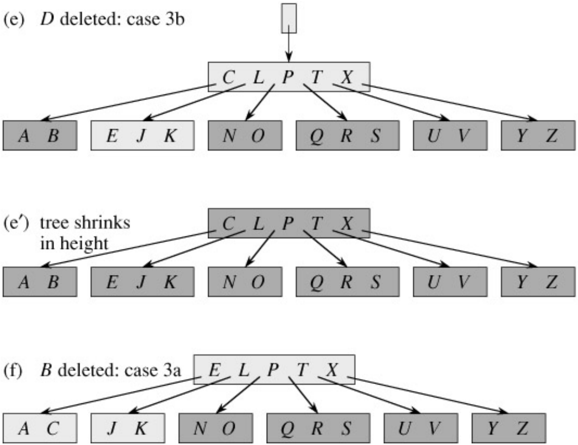
[Cormen, Leiserson, Rivest, and Stein. *Introduction to Algorithms*, 2nd ed. MIT Press, 2001.]

Deletion examples



[Cormen, Leiserson, Rivest, and Stein. *Introduction to Algorithms*, 2nd ed. MIT Press, 2001.]

Deletion examples



[Cormen, Leiserson, Rivest, and Stein. *Introduction to Algorithms*, 2nd ed. MIT Press, 2001.]

Priority queue

- Abstract data structure ordering elements by priority
- Elements enqueued with priority, dequeued in order of highest priority
- Common implementations: heap or binary search tree
- Operations: insertion, peek/extract max-priority element, increase element priority

Heaps

- Heap - tree with heap-ordering property: priority(child) $\leq$ priority(parent)
- More sophisticated heaps exist – e.g. binomial heap, Fibonacci heap
- We'll focus on simple binary heaps
- Usually implemented as an array with top element at beginning
- Can sort data using a heap – O$(n \log n)$ worst case in-place sort!

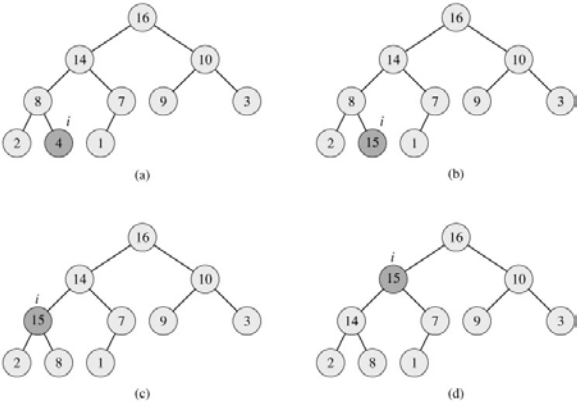
Extracting data

- Heap-ordering property $\Rightarrow$ maximum priority element at top of heap
- Can peek by looking at top element
- Can remove top element, move last element to top, and swap top element down with its children until it satisfies heap-ordering property:
  1. start at top
  2. find largest of element and left and right child; if element is largest, we are done
  3. otherwise, swap element with largest child and repeat with element in new position

Inserting data/increasing priority

- Insert element at end of heap, set to lowest priority $-\infty$
- Increase priority of element to real priority:
    1. start at element
    2. if new priority less than parent's, we are done
    3. otherwise, swap element with parent and repeat

Example of inserting data



[Cormen, Leiserson, Rivest, and Stein. *Introduction to Algorithms*, 2nd ed. MIT Press, 2001.]

Summary

Topics covered:

- Using external libraries
  - symbols and linkage
  - static vs. dynamic linkage
  - linking to your code
  - symbol clashing
- Creating libraries
- Data structures
  - B-tree
  - priority queue

6.087 Practical Programming in C
January (IAP) 2010