Data Structures

Linked Lists

two kinds of linked list structures

C lists (usually have links embedded into data payload)

definition in C

```
struct node {
  int x;
  struct node *next;
};
```

Pros

- links are alway there
- no need to allocate `cons' nodes when adding element to a list
- more efficient when adding, removing element to lists
- easier to manage when resource management facility is poor

Cons

- This gets worse when a datum can be in multiple lists
- The pointer will always take space even the datum is not in the list
- redo all the link following code in each loop
- no abstraction
- hurt modularity, reduce reuse of code

final word

- trade space for time
- see Linux task_struct
- often used in operating systems

Lisp lists (have special "cons nodes" that link to each other, but only pointing to the payload data)

definition in Haskell

```
data List a = Nil | Cons a (List a)
```

definition in Lisp (Typed Racket)

```
(struct List ((datum : Any)
              (next  : List)))
```

Pros

- abstraction, generic programming
- clean and reusable code for list operations

Cons

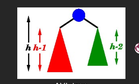
- allocation and deallocation of `cons' cells may take time

final word

- use them when we can afford a little more running time

Search Trees

AVL tree



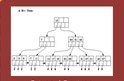
balanced trees

video example

splay trees

- rotate the most frequently used element to top
- no need to keep perfectly balanced
- simple to implement
- adapt to dynamic data

B tree / B+ tree



- generalization of "binary" search trees
- use large branching factor to reduce tree height
- B+tree vs B-tree
  - B+-tree stores all data in leaf nodes
  - interior nodes only store "key dividers"
- may be combined with "splay" to produce B+/splay tree

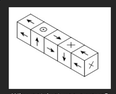
Purely Functional Data Structures (aka persistent data structures)

Purely Functional Data Structures (by Chris Okasak) books

- If used ... rationale
  - sequentially
    - behaves like *prepaid* facilities
    - Pros
      - no need to revert the effects at backtracking
      - because "old version" was kept intact
      - just need to get back to the original handle
      - easier to manage
    - Cons
      - won't save running time
      - won't waste running time
      - for building augmented structure
      - time
      - space
      - must *prepay*
      - *peak space usage* can be much larger than imperative data structures when the recursion gets deep
  - concurrently
    - seems to be the only correct way
    - because concurrent processes must access different versions of the data at the same time, side-effects will cause *inadvertent communications* which may cause *race conditions*
- examples
  - Lisp's lists
    - when not using set-carl or set-cdrl
    - only use cons, car, cdr
  - persistent search trees
    - copy every node on the path until the root
    - creates a new root

Hash Tables

- A: The key space may be huge
  - not enough (address) space for all keys
  - Observation: keys in huge spaces may appear only sparsely in practice
    - This is when hash tables work well
- Why not just use arrays? (constant time look up)



- Solution: distribute and group some keys together. Put them into the same slot
  - use hash function to "distribute" the key space
    - takes some luck to get a good distribution
      - Murmur hash
    - probablematic cache locality
      - hash value of adjacent elements may be far apart
      - but cache stores contents of memory at nearby locations
  - put the "hashed key" into an array
    - there may be conflicts
    - how to deal with conflicts
      - many of the ways in textbooks don't really work well
      - put conflicting elements into a list
      - use extendible hash

Can hash tables also be trees?

- "hashed key space" may be segmented (constructing a tree-like structure)
  - most significant "bits" on upper level nodes
    - refine the selection on lower nodes using lower bits
    - If all bits are used at the same time, it becomes an array
  - branching factor is proportional to the size of key space
    - "constant search time" is only achieved when the table size is not much smaller than key space size
      - can probably achieve the same with B+ trees
      - huge branching factor leads to small tree height

extendible hash

- when too many things gets into the same bucket, split the bucket
- goal: split the key space of a tree node into two
- This is much like a page table