Algorithms and Data Structures (II)

Course 2,
Gabriel Istrate

March 9, 2023

First of all ...

Example: Selection sort vs. HEAPSORT

Selection SORT

- Elements to sort in a vector.
- Find maximum element.
- Swap it with the last element.
- Proceed recursively.

Complexity

- Finding maximum in a vector: $\Theta(n)$.
- Complexity analysis: $T(n) = T(n-1) + \theta(n)$.
- Conclusion: $\Theta(n^2)$

Example: Selection sort vs. HEAPSORT

Selection SORT

- Bottleneck in Selection Sort: Find maximum element
- If we could improve finding max

HEAPSORT

- Algorithm: Same idea.
- Bottleneck: FindMax $\theta(n)$. Replace it with $O(1)$ operation (via heaps).
- Complexity $W(n) = \theta(\log n)$ (need also to update heap via HEAPIFY)
- New algorithm: complexity $\theta(n \log(n))$.

Why data structures ?

- Data structures: algorithm development via primitive operations.

Modularity in algorithm design

You don't build a house from scratch (bricks, frames, drywalls). Same with algorithms/code.

- Easier to solve problem/test solution only once.
- Correctness: easier to check. easier to update.

Why data structures ? Performance.

- You google something. Don't want to wait 100 seconds ! Search: fast.
- You play a game. Game engine must quickly retrieve/update objects you see in front of you when you move your viewport.
- Operations: often abstracted from requirements.

Most frequent operations should be fast.

How do you measure performance ?

$O(n \log n)$, $\Theta(\log n)$ ...

Example (operations from requirements)

$$\underbrace{1\quad 2\quad 3}\ \ \underbrace{7\quad 5\quad 6\quad 4}\ \ \underbrace{8\quad 9\quad 10}\ \ \underbrace{14\quad 13\quad 12\quad 11}\ldots$$

- TCP: basis for much of Internet traffic.
- Data requirement: We need to buffer a packet that is out-of order.
- We need to pop elements that become in-order.
- We need to test emptiness of buffer.
- We need to produce first missing element (ACK).
- Operation performance O(1)?.

Concepts

- A data structure is a way to organize and store information
  - to facilitate access, or for other purposes
- A data structure has an interface consisting of procedures for adding, deleting, accessing, reorganizing, etc.
- A data structure stores data and possibly meta-data
  - e.g., a heap needs an array A to store the keys, plus a variable A.heap-size to remember how many elements are in the heap

What are data structures more concretely ?

- data ...
- E.g. complex numbers: two floats.
- ... together with operations one can perform on the data ...
  Example: integer + (addition), - (subtraction), $\cdot$ (multiplication).
- ... and performance guarantees.

Note !

How to precisely implement operations is not a part of data structure specification. Concepts, not code.

Data types

- All DS that share a common structure and expose the same set of operations.
- Predefined data types: array, structures, files.
- Scalar data type: ordering relation exists among elements.
- More complicated: dynamic DS. Lists, circular lists, trees, hash tables, graphs.

C++: Standard template library (STL):

library of container classes, algorithms, and iterators; provides many of the basic algorithms and data structures of computer science

Example: Array data type/Vector

- Ensures random access to its elements.
- Complexity O(1).
- Composed of objects of the same type.

Implementations

- int myarray[10]; One dimensional arrays.
- Multidimensional arrays.
  type name[lim_1]...[lim_n];
- implementation in C++/STL: vector.

Example using vector class

```
#include<vector>
        using namespace std;

        int main(){

        static const int SIZE = 10000;
        vector<int> arr( SIZE );
        arr.append(125);
        ....
        }
```

Vectors the C++/data structures way

- Vector: black-box.
- Random access: arr[i] should take $\Theta(1)$ time.
- Black box (class implementation) may implement some other operations, e.g. append.

Main point

You didn't implement vector yourself. All you care is what operations can you execute, and how complex they are.

This course

Define, implement various "data structures", and use them to get better algorithms.

Some minimal C/C++ recap

You have an entire course for more.

Pointers in C(++)

Variables that hold addresses of other variables.

int i=15,j, *p,*q;

Dynamic memory allocation: p= new int;
Assignment: *p=20;
Deallocation: delete p;
Dangling reference: upon deallocation should assign p = 0;

Pointers and arrays

```
int a[5],*p;

for(sum=a[0],i=1;i<5;i++)
 sum += a[i];
or
for (sum=*a,i=1;i<5;i++)
 sum += *(a+i);
or
for(sum=*a,p=a+1;p<a+5;p++)
 sum += *p;

p = new int[n];
delete [] p;
```

Pointers and reference variables

int $n = 5, *p = \&n, \&r = n;$.
r is a reference variable. Must be initialized in definition as reference to a particular variable.

reference: different name for/constant pointer to variable.
cout « n « ' '« *p«' '« r« endl;
5 5 5

n= 7 (*p = 7, r = 7)
cout « n « ' '« *p«' '« r« endl;
7 7 7

cout: C++ way to print. BEST WAY TO PASS PARAMETER: const reference variables;

C++: classes, objects, member functions, oh my !

- in C++: classes - user-defined data types.
- objects: instantiations of classes.
- objects have behavior, member functions.

Example

- Assume dog is a C++ class.
- Assume Buddy is an "object" of type dog.
- Dogs behavior: bark, member function with no parameters.
- To make Buddy bark: Buddy.bark() call member function bark that belongs to Buddy.
- C++: only so-called public member functions can be called from outside the class code.

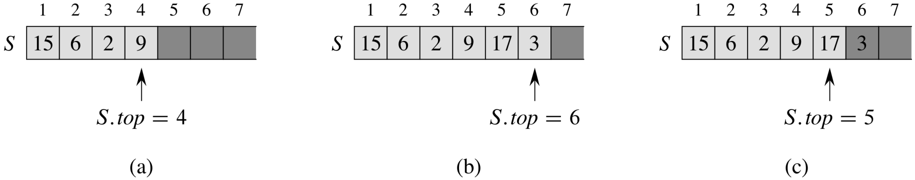
So let's start ...

Today (time permitting):

- Stacks
- Queues
- Dequeues
- Linked Lists
- Skip Lists.

- A Stack is a sequential organization of items in which the last element inserted is the first element removed. They are often referred to as LIFO, which stands for "last in first out."
- Examples: letter basket, stack of trays, stack of plates.
- Only element that may be accessed: the one that was most recently inserted.
- There are only two basic operations on stacks, the push (insert), and the pop (read and delete).

Stacks: Implementation



- (a). Stack representing set S $= \{2, 6, 9, 15\}$.
- (b). After PUSH(S,3).
- (c). After POP(S).

Operator Precedence Parsing

- We can use the stack class we just defined to parse and evaluate mathematical expressions like:

$$5 * (((9 + 8) * (4 * 6)) + 7)$$

- First, we transform it to postfix notation:

$$5\ 9\ 8\ +\ 4\ 6\ *\ *\ 7\ +\ *$$

- Usual form for arithmetic expressions: infix. term1 op term2.
- Postfix notation: term1 term2 op.
- How to convert infix to postfix: later !

Then, the following C++ routine uses a stack to perform this evaluation:

```
char c;
Stack acc(50);
int x;
while (cin.get(c))
{
x = 0;
while (c == ' ' ) cin.get(c);
if (c == '+') x = acc.pop() + acc.pop();
if (c == '*') x = acc.pop() * acc.pop();
while (c≥ '0' && c ≤ '9')
x = 10*x + (c-'0'); cin.get(c);
acc.push(x);
}
cout << acc.pop();
```

Explanation of code

- We read one character at a time in c.
- In x we compute the value of the currently evaluated expression.
- After computing it we push the value on the stack - we will need it later.
- When reading an op we take the last two value off the stack and apply the op on them and assign this to x.
- When reading a digit we update value of x by making the last read digit the least significant one.

Stacks: Applications

- Algorithms (later).
- Recursion removal.
- Reversing things.
- Procedure call and procedure return is similar to matching symbols:
  - When a procedure returns, it returns to the most recently active procedure.
  - When a procedure call is made, save current state on the stack. On return, restore the state by popping the stack.
  - Formal languages: pushdown automata.

Queues

- The ubiquitous “first-in first-out” container (FIFO)

Queues

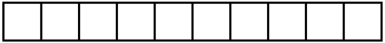
- The ubiquitous "first-in first-out" container (FIFO)
- Interface
  - Enqueue(Q, x) adds element x at the back of queue Q
  - Dequeue(Q) extracts the element at the head of queue Q

Queues

- The ubiquitous "first-in first-out" container (FIFO)
- Interface
  - $\text{Enqueue}(Q, x)$ adds element x at the back of queue Q
  - $\text{Dequeue}(Q)$ extracts the element at the head of queue Q
- Implementation
  - Q is an array of fixed length $Q.\text{length}$
    - i.e., Q holds at most $Q.\text{length}$ elements
    - enqueueing more than Q elements causes an "overflow" error
  - $Q.\text{head}$ is the position of the "head" of the queue
  - $Q.\text{tail}$ is the first empty position at the tail of the queue

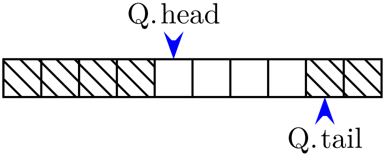
Enqueue

```
Enqueue(Q,x)
if Q.queue-full
        error "overflow"
else Q[Q.tail] = x
        if Q.tail < Q.length
                Q.tail = Q.tail + 1
        else Q.tail = 1
        if Q.tail == Q.head
                Q.queue-full = true
        Q.queue-empty = false
```



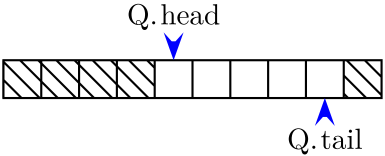
Enqueue

```
Enqueue(Q,x)
if Q.queue-full
        error "overflow"
else Q[Q.tail] = x
        if Q.tail < Q.length
                Q.tail = Q.tail + 1
        else Q.tail = 1
        if Q.tail == Q.head
                Q.queue-full = true
        Q.queue-empty = false
```



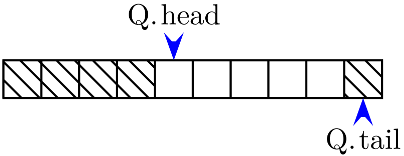
Enqueue

```
Enqueue(Q,x)
if Q.queue-full
    error "overflow"
else Q[Q.tail] = x
    if Q.tail < Q.length
        Q.tail = Q.tail + 1
    else Q.tail = 1
    if Q.tail == Q.head
        Q.queue-full = true
    Q.queue-empty = false
```



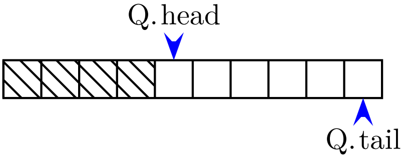
Enqueue

```
Enqueue(Q,x)
if Q.queue-full
        error "overflow"
else Q[Q.tail] = x
        if Q.tail < Q.length
                Q.tail = Q.tail + 1
        else Q.tail = 1
        if Q.tail == Q.head
                Q.queue-full = true
        Q.queue-empty = false
```



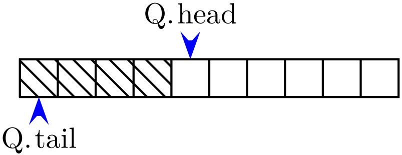
Enqueue

```
Enqueue(Q,x)
if Q.queue-full
    error "overflow"
else Q[Q.tail] = x
    if Q.tail < Q.length
        Q.tail = Q.tail + 1
    else Q.tail = 1
    if Q.tail == Q.head
        Q.queue-full = true
    Q.queue-empty = false
```



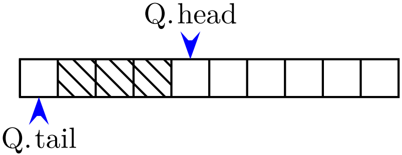
Enqueue

```
Enqueue(Q,x)
if Q.queue-full
        error "overflow"
else Q[Q.tail] = x
        if Q.tail < Q.length
                Q.tail = Q.tail + 1
        else Q.tail = 1
        if Q.tail == Q.head
                Q.queue-full = true
        Q.queue-empty = false
```



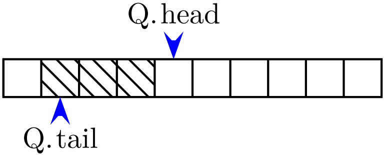
Enqueue

```
Enqueue(Q,x)
if Q.queue-full
        error "overflow"
else Q[Q.tail] = x
        if Q.tail < Q.length
                Q.tail = Q.tail + 1
        else Q.tail = 1
        if Q.tail == Q.head
                Q.queue-full = true
        Q.queue-empty = false
```



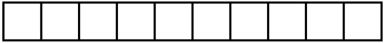
Enqueue

```
Enqueue(Q,x)
if Q.queue-full
      error "overflow"
else Q[Q.tail] = x
      if Q.tail < Q.length
            Q.tail = Q.tail + 1
      else Q.tail = 1
      if Q.tail == Q.head
            Q.queue-full = true
      Q.queue-empty = false
```



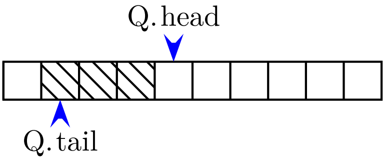
Dequeue

```
Dequeue(Q)
if Q.queue-empty
        error "underflow"
else x = Q[Q.head]
        if Q.head < Q.length
                Q.head = Q.head + 1
        else Q.head = 1
        if Q.tail == Q.head
                Q.queue-empty = true
        Q.queue-full = false
        return x
```



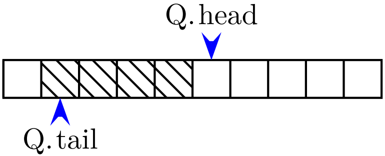
Dequeue

```
Dequeue(Q)
if Q.queue-empty
    error "underflow"
else x = Q[Q.head]
    if Q.head < Q.length
        Q.head = Q.head + 1
    else Q.head = 1
    if Q.tail == Q.head
        Q.queue-empty = true
    Q.queue-full = false
    return x
```

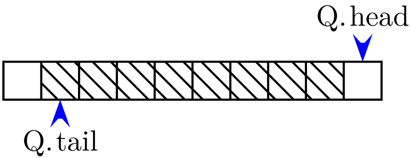


Dequeue

```
Dequeue(Q)
if Q.queue-empty
    error "underflow"
else x = Q[Q.head]
    if Q.head < Q.length
        Q.head = Q.head + 1
    else Q.head = 1
    if Q.tail == Q.head
        Q.queue-empty = true
    Q.queue-full = false
    return x
```

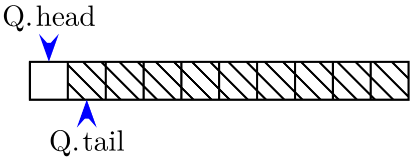


```
Dequeue(Q)
if Q.queue-empty
    error "underflow"
else x = Q[Q.head]
    if Q.head < Q.length
        Q.head = Q.head + 1
    else Q.head = 1
    if Q.tail == Q.head
        Q.queue-empty = true
    Q.queue-full = false
    return x
```



Dequeue

```
Dequeue(Q)
if Q.queue-empty
      error "underflow"
else x = Q[Q.head]
      if Q.head < Q.length
            Q.head = Q.head + 1
      else Q.head = 1
      if Q.tail == Q.head
            Q.queue-empty = true
      Q.queue-full = false
      return x
```



Applications of Queues

- Scheduling (disk, CPU)
- Used by operating systems to handle congestion.
- Algorithms (we'll see): breadth-first search.

Stacks,Queues: Scorecard

Algorithm	Complexity
Stack-Empty	O(1) ✓
Push	O(1) ✓
Pop	O(1) ✓
Enqueue	O(1) ✓
Dequeue	O(1) ✓
Restrictions:	LIFO/FIFO orders only. ×

Deques

- Like queues but can enqueue/dequeue at both ends.
- Can modify the code for queues, add two more procedure.
- do it !
- Complexity scorecard: similar to queues.

Dynamic sets

Major problem this semester:

Represent a set S whose elements may vary through time. May want to perform some of:

- INSERT(S,x)
- DELETE(S,x)
- SEARCH(S,x). Result YES/NO. Better: handle for x, if found.
- MIN(S)
- MAX(S)
- SUCC(S,x), PRED(S,x)

Example: stacks/queues

- Stacks: dynamic sets with LIFO order.
- Queues: dynamic sets with FIFO order.

Dictionary

- A dictionary is an abstract data structure that represents a set of elements (or keys)
  - a dynamic set

Dictionary

- A dictionary is an abstract data structure that represents a set of elements (or keys)
  - a dynamic set
- Interface (generic interface)
  - $\mathrm{Insert}(\mathrm{D}, \mathrm{k})$ adds a key k to the dictionary D
  - $\mathrm{Delete}(\mathrm{D}, \mathrm{k})$ removes key k from D
  - $\mathrm{Search}(\mathrm{D}, \mathrm{k})$ tells whether D contains a key k

Dictionary

- A dictionary is an abstract data structure that represents a set of elements (or keys)
  - a dynamic set
- Interface (generic interface)
  - $\mathrm{Insert}(D, k)$ adds a key k to the dictionary D
  - $\mathrm{Delete}(D, k)$ removes key k from D
  - $\mathrm{Search}(D, k)$ tells whether D contains a key k
- Implementation
  - many (concrete) data structures

Dictionary

- A dictionary is an abstract data structure that represents a set of elements (or keys)
  - a dynamic set
- Interface (generic interface)
  - $\mathrm{Insert}(D, k)$ adds a key k to the dictionary D
  - $\mathrm{Delete}(D, k)$ removes key k from D
  - $\mathrm{Search}(D, k)$ tells whether D contains a key k
- Implementation
  - many (concrete) data structures
  - we'll see: hash tables

Direct-Address Table

- A direct-address table implements a dictionary

Direct-Address Table

- A direct-address table implements a dictionary
- The universe of keys is $U = \{1, 2, \ldots, M\}$

Direct-Address Table

- A direct-address table implements a dictionary
- The universe of keys is $U = \{1, 2, \ldots, M\}$
- Implementation
  - an array T of size M
  - each key has its own position in T

Direct-Address Table

- A direct-address table implements a dictionary
- The universe of keys is $U = \{1, 2, \dots, M\}$
- Implementation
  - an array T of size M
  - each key has its own position in T

```
Direct-Address-Insert(T, k)
1  T[k] = true
```

```
Direct-Address-Delete(T, k)
1  T[k] = false
```

```
Direct-Address-Search(T, k)
1  return T[k]
```

Direct-Address Table (2)

- Complexity

Direct-Address Table (2)

- Complexity

All direct-address table operations are $O(1)$✓

Direct-Address Table (2)

- Complexity

All direct-address table operations are $O(1)$ ✓

So why isn't every set implemented with a direct-address table?

Direct-Address Table (2)

- Complexity

All direct-address table operations are $O(1)$✓

So why isn't every set implemented with a direct-address table?

- Space complexity is $\Theta(|U|)$×
  - $|U|$ is typically a very large number—U is the universe of keys!
  - the represented set is typically much smaller than $|U|$
    - i.e., a direct-address table usually wastes a lot of space

Direct-Address Table (2)

- Complexity

  All direct-address table operations are $O(1)$✓

  So why isn't every set implemented with a direct-address table?

- Space complexity is $\Theta(|U|)$×
  - $|U|$ is typically a very large number—U is the universe of keys!
  - the represented set is typically much smaller than $|U|$
    - i.e., a direct-address table usually wastes a lot of space

- Want: the benefits of a direct-address table but with a table of reasonable size.

Direct Access Tables: Scorecard

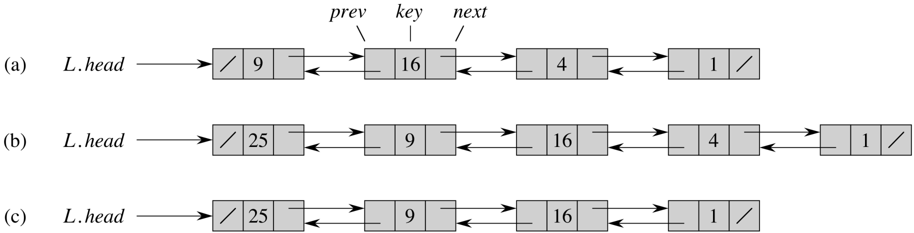
Algorithm	Complexity
INSERT	O(1)✓
DELETE	O(1)✓
SEARCH	O(1)✓
MEMORY:	θ(M)×

- Interface
  - List-Insert(L, x) adds element x at beginning of a list L
  - List-Delete(L, x) removes element x from a list L
  - List-Search(L, k) finds an element whose key is k in a list L

Linked Lists

- Interface
  - List-Insert(L, x) adds element x at beginning of a list L
  - List-Delete(L, x) removes element x from a list L
  - List-Search(L, k) finds an element whose key is k in a list L
- Implementation
  - a doubly-linked list
  - each element x: two "links" $x.prev$ and $x.next$ to the previous and next elements, respectively
  - each element x: key $x.key$

Linked List: Implementation



- (a). Linked list representing set $S = \{1, 4, 9, 16\}$.
- (b). After LIST-INSERT(S,25).
- (c). After LIST-DELETE(S,4).

Linked List: Implementation

```
List-Init(L)
    L.head = NIL
```

```
List-Insert(L, x)
    x.next = L.head
    if L.head ≠ NIL
        L.head.prev = x
    L.head = x
    x.prev = NIL
```

```
List-Search(L, k)
    x = L.head.next
    while x ≠ NIL ∧ x.key ≠ k
        x = x.next
    return x
```

Linked List: Implementation (II)

```
List-Delete(L, x)
if x.prev ≠ NIL
    x.prev.next = x.next
else L.head = x.next
if x.next ≠ NIL
    x.next.prev = x.prev
```

Linked List With a "Sentinel"

- instead of NIL sometimes convenient to have a dummy "sentinel" element L. nil
- Simplifies LIST-DELETE .
- Adds more memory $\times$.

Linked List With a "Sentinel"

```
List-Init(L)
L.nil.prev = L.nil
L.nil.next = L.nil
```

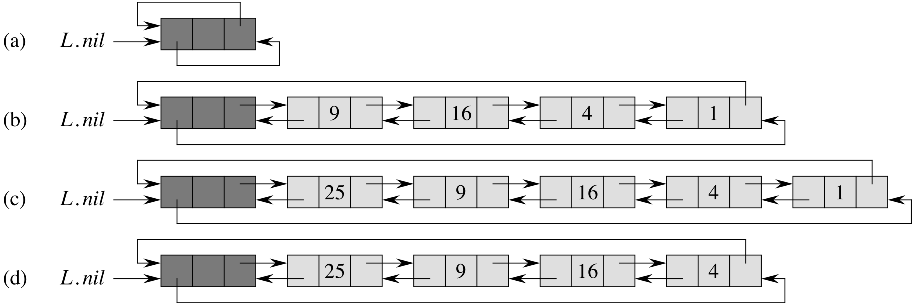
```
List-Insert(L, x)
x.next = L.nil.next
L.nil.next.prev = x
L.nil.next = x
x.prev = L.nil
```

```
List-Search(L, k)
x = L.nil.next
while x ≠ L.nil ∧ x.key ≠ k
    x = x.next
return x
```

Linked Lists: Observations on Implementation

- Insert: at the head of the list.
- Possible: insert arbitrary position.

Circular Linked Lists



- Can use nil sentinel as head of the list.
- (a): empty circular list.
- (b): Linked list representing set $S = \{1, 4, 9, 16\}$.
- (c): After LIST-INSERT(S,25).
- (d): After LIST-DELETE(S,4).

Linked Lists: Scorecard

Linked Lists: Scorecard

Algorithm	Complexity
List-Insert	

Linked Lists: Scorecard

Algorithm	Complexity
List-Insert	O(1) ✓
List-Delete (with pointer)	

Linked Lists: Scorecard

Algorithm	Complexity
List-Insert	O(1) ✓
List-Delete (with pointer)	O(1) ✓
List-Search	

Linked Lists: Scorecard

Algorithm	Complexity
List-Insert	O(1) ✓
List-Delete (with pointer)	O(1) ✓
List-Search	Θ(n) ×

Linked Lists: to conclude

- Can reimplement Stacks/Queues using Linked Lists.
- Implementation with pointers: will not pass the class if you don't know it !

Advanced topic - Skip lists

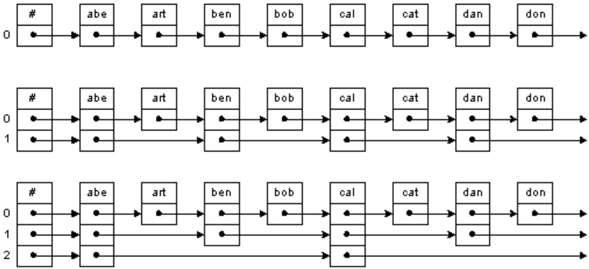
Caution

Topic not in Cormen. See Drozdek for details/C++ implementation.

- Problem with linked list: search is slow !... even when elements sorted.
- Solution: lists of ordered elements that allow skipping some elements to speed up search.
- Skip lists: variant of ordered linked lists that makes such search possible.

More advanced data structure (W. Pugh "Skip lists: a Probabilistic Alternative to Balanced Trees", Communication of the ACM 33(1990), pp. 668-676.) If anyone curious/interested in data structures/algorithms, can give paper to read; taste how a research article looks like.

Skip lists



Where does this ever get applied ?

...

Skip lists in real life

According to Wikipedia:

- MemSQL - skip lists as prime indexing structure for its database technology.
- Cyrus IMAP server - "skiplist" backend DB implementation
- Lucene uses skip lists to search delta-encoded posting lists in logarithmic time.
- QMap (up to Qt 4) template class of Qt that provides a dictionary.
- Redis, ANSI-C open-source persistent key/value store for Posix systems, skip lists in implementation of ordered sets.
- nessDB, a very fast key-value embedded Database Storage Engine.
- skipdb: open-source DB format using ordered key/value pairs.
- ConcurrentSkipListSet and ConcurrentSkipListMap in the Java 1.6 API.

Skip lists in real life (II)

According to Wikipedia:

- Speed Tables: fast key-value datastore for Tcl that use skiplists for indexes and lockless shared memory.
- leveldb, a fast key-value storage library written at Google that provides an ordered mapping from string keys to string values
- MuQSS Scheduler for the Linux kernel uses skip lists
- SkipMap uses skip lists as base data structure to build a more complex 3D Sparse Grid for Robot Mapping systems.

Skip lists: implementation

What we want

$k = 1, \ldots, \lfloor \log_2(n) \rfloor$, $1 \le i \le \lfloor n/2^{k-1} \rfloor - 1$.

- Item $2^{k-1} \cdot i$ points to item $2^{k-1} \cdot (i+1)$.
- every second node points to positions two node ahead,
- every fourth node points to positions four nodes ahead,
- every eigth node points to positions eigth nodes ahead,
- ......, and so on.

- Different number of pointers in different nodes in the list !
- half the nodes only one pointer.
- a quarter of the nodes two pointers,
- an eigth of the nodes four pointers,
- ......, and so on.
- $n \log_2(n)/2$ pointers.

Search Algorithm

1. First follow pointers on the highest level until a larger element is found or the list is exhausted.
2. If a larger element is found, restart search from its predecessor, this time on a lower level.
3. Continue doing this until element found, or you reach the first level and a larger element or the end of the list.

Inserting and deleting nodes

Major problem

- When inserting/deleting a node, pointers of prev/next nodes have to be restructured.
- Solution: rather than equal spacing, random spacing on a level.
- Invariant: Number of nodes on each level: equal, in expectation to what it would be under equal spacing

Principle

If you're traveling 10 meters in 10 steps, a step is on average one meter.

Inserting and deleting nodes (II)

- Level numbering: start with zero.
- New node inserted: probability 1/2 on first level, 1/4 second level, 1/8 third level, ..., etc.
- Function chooseLevel: chooses randomly the level of the new node.
- Generate random number. If in [0,1/2] level 1, [1/2,3/4] level 2, etc.
- To delete node: have to update all links.

Indexing

Computing the i'th element faster than in O(i)

- If we record "step sizes" in our lists we can even mimic indexing !
- Start on highest level.
- If step too big, restart search from predecessor, this time on a lower level.
- Continue doing this until element found.

Update "step sizes" by insertion/deletion

Easy if you have doubly linked lists.

- On deletion: pred[i].size$+=$ deleted.size on all levels i.
- On insertion: Simply keep track of predecessors and index of the inserteed sequence.

Skip Lists: Scorecard

Method	Average	Worst-Case
SPACE: ✓	O(n)	O(nlog(n))
SEARCH: ✓	O(log(n))	O(n)
INSERT: ✓	O(log(n))	O(n)
DELETE: ✓	O(log(n))	O(n)

- quite practical ! ✓
- Probabilistic, worst-case still bad. ×
- Not completely easy to implement. ×.

Compared to what ?

Binary search trees. Will learn about them later.