CMSC 341

Skip Lists

Looking Back at Sorted Lists

Sorted Linked List

What is the worst case performance of find( ), insert( )?

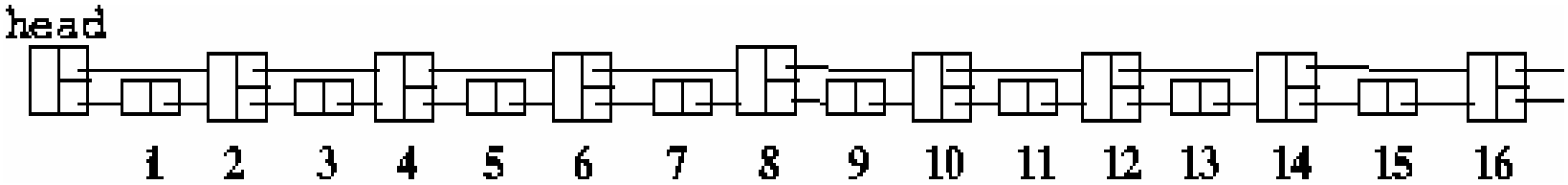
Sorted Array

What is the worst case performance of find( ), insert( )?

An Alternative Sorted Linked List

What if you skip every other node?

- Every other node has a pointer to the next and the one after that



Find :

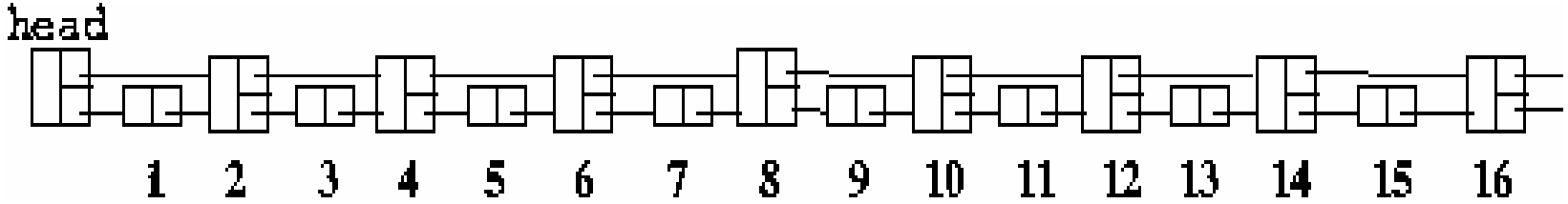
- follow “skip” pointer until target < this.skip.element

Resources

- Additional storage

Performance of find( )?

Skipping every 2nd Node

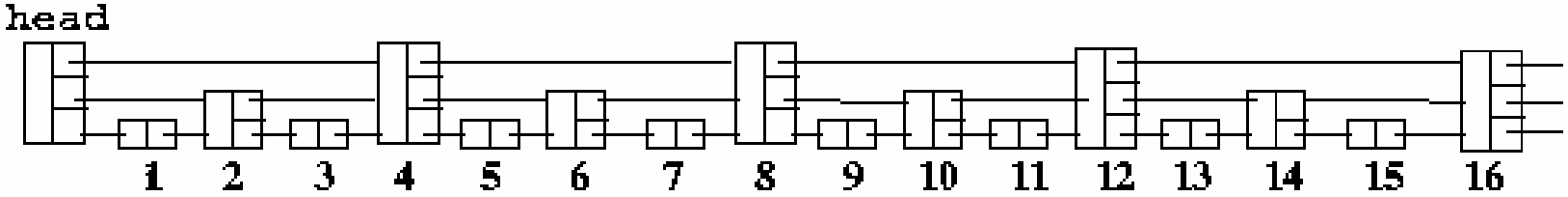


The value stored in each node is shown below the node and corresponds to the the position of the node in the list.

It's clear that find( ) does not need to examine every node. It can skip over every other node, then do a final examination at the end. The number of nodes examined is no more than $\lceil n/2 \rceil + 1$.

For example the nodes examined finding the value 15 would be 2, 4, 6, 8, 10, 12, 14, 16, 15 -- a total of $\lceil 16/2 \rceil + 1 = 9$.

Skipping every 2nd and 4th Node



The find operation can now make bigger skips than the previous example. Every 4th node is skipped until the search is confined between two nodes of size 3. At this point as many as three nodes may need to be scanned. It's also possible that some nodes may be examined more than once. The number of nodes examined is no more than $\lceil n / 4 \rceil + 3$.

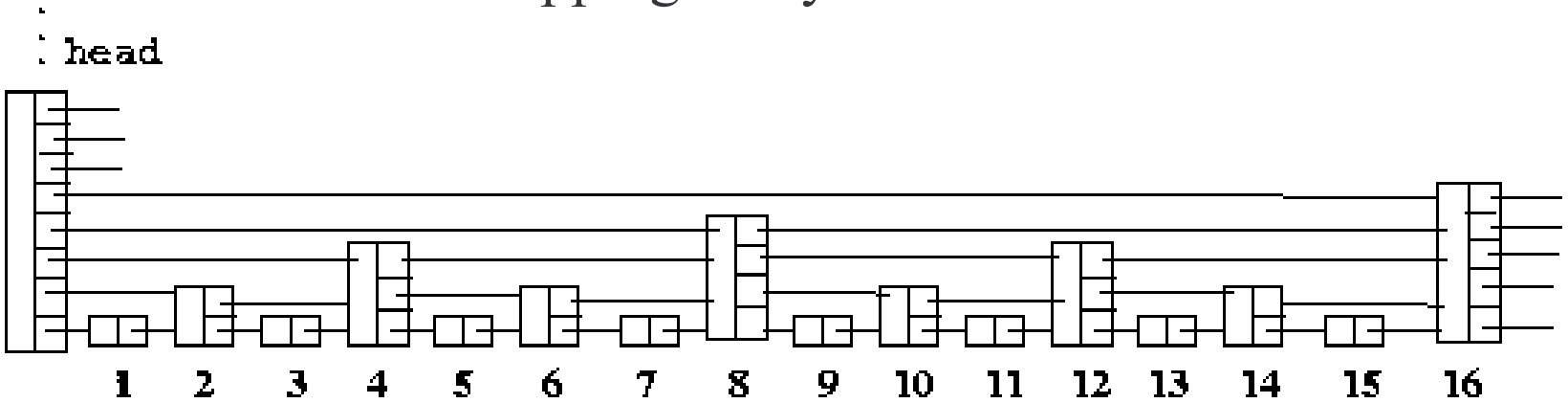
Again, look at the nodes examined when searching for 15

New and Improved Alternative

Add hierarchy of skip pointers

- every 2^i-th node points 2^i nodes ahead
- For example, every 2^{nd} node has a reference 2 nodes ahead; every 8^{th} node has a reference 8 nodes ahead

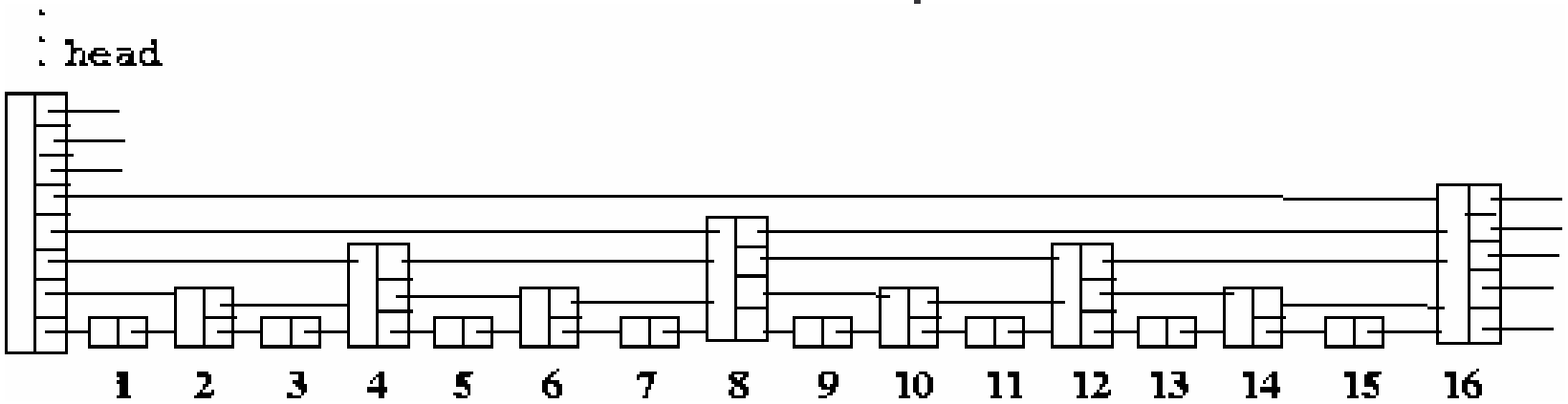
Skipping every 2^i-th node



Suppose this list contained 32 nodes and we want to search for some value in it. Working down from the top, we first look at node 16 and have cut the search in half. When we look again one level down in either the right or left half, we have cut the search in half again. We continue in this manner until we find the node being sought (or not).

This is just like binary search in an array. Intuitively we can understand why the max number of nodes examined is O(lg N).

Some serious problems



This structure looks pretty good, but what happens when we insert or remove a value from the list? Reorganizing the the list is O(N).

For example, suppose the first element of the list was removed. Since it's necessary to maintain the strict pattern of node sizes, it's easiest to move all the values toward the head and remove the end node. A similar situation occurs when a new node is added

Skip Lists

Concept:

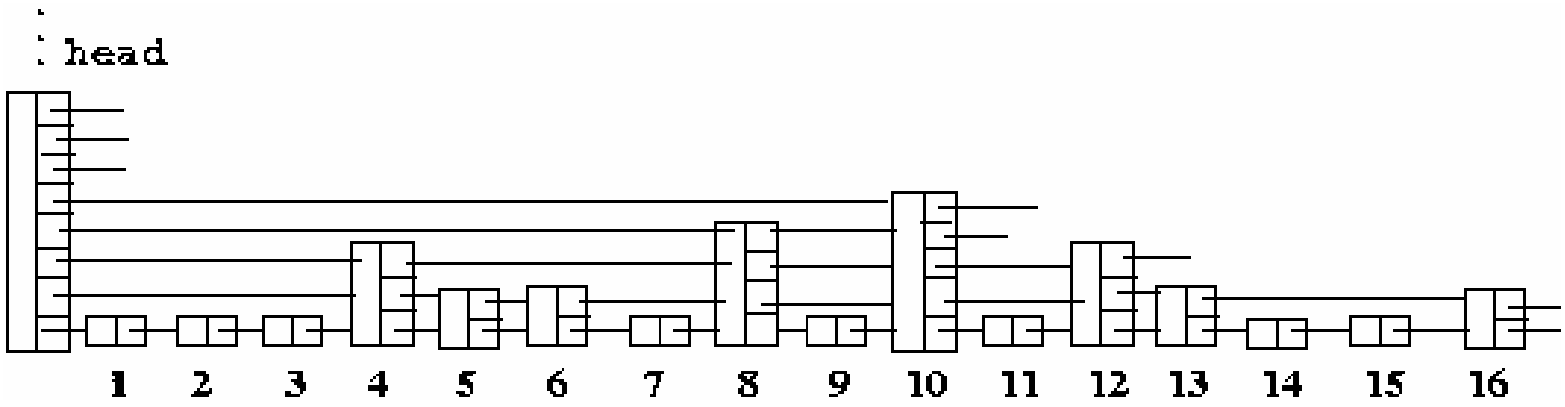
A skip list maintains the same **distribution** of nodes, but without the requirement for the rigid pattern of node sizes

- 1/2 have 1 pointer
- 1/4 have 2 pointers
- 1/8 have 3 pointers
- …
- $1/2^i$ have i pointers

It's no longer necessary to maintain the rigid pattern by moving values around for insert and remove. This gives us a ***high probability*** of still having O(lg N) performance. The probability that a skip list will behave badly is very small.

The number of forward reference pointers a node has is its "size"

A Probabilistic Skip List



The distribution of node sizes is exactly the same as the previous figure, the nodes just occur in a different pattern.

Inserting a node

When inserting a new node, we choose the size of the node probabilistically.

Every skip list has an associated (and fixed) probability, ***p***, that determines the distribution of node sizes. A fraction, ***p***, of the nodes that have at least ***r*** forward references also have ***r*** + ***1*** forward references.

Skip List Insert

To insert node:

- create new node with random size
- for each pointer, i , connect to next node with at least i pointers

```
int GenerateNodeSize(double p, int maxSize)
{
   int size = 1;
   while (drand48() < p) size++;
   return (size >maxSize) ? maxSize : size;
}
```

An aside on Node Distribution

Given an infinitely long skip list with associated probability ***p***, it can be shown that **1 – p** nodes will have just one forward reference.

This means that **p(1 – p)** nodes will have exactly two forward references and in general $\mathbf{p^k(1 - p)}$ nodes will have **k + 1** forward reference pointers.

For example, with p = 0.5

0.5 (1/2) of the nodes will have exactly one forward reference

0.5 (1 – 0.5) = 0.25 (1/4) of the nodes will have 2 references

0.5^2 (1 – 0.5) = 0.125 (1/8) of the nodes will have 3 references

0.5^3 (1 – 0.5) = 0.0625 (1/16) of the nodes will have 4 references

Work out the distribution for p = 0.25 (1/4) for yourself.

Determining the size of the Header Node

The size of the header node (the number of forward references it has) is the maximum size of any node in the skip list and is chosen when the empty skip list is constructed (i.e. it must be predetermined)

Dr. Pugh has shown that the maximum size should be chosen as $\log_{1/p} N$. For $p = ½$, the maximum size for a skip list with 65,536 elements should be no smaller than $\log_2 65536 = 16$.

Performance considerations

The ***expected*** time to find an element (and therefore to insert or remove) is O(lg N). It is possible for the time to be substantially longer if the configuration of nodes is unfavorable for a particular operation. Since the node sizes are chosen randomly, it is possible to get a "bad" run of sizes. For example, it is possible that each node will be generated with the same size, producing the equivalent of an ordinary linked list. A "bad" run of sizes will be less important in a long skip list than in a short one. The probability of poor performance decreases rapidly as the number of nodes increases.

More performance

The probability that an operation takes longer than expected is function of the associated probability ***p***. Dr. Pugh calculated that with p = 0.5 and 4096 elements, the probability that the actual time will exceed the expected time by more than a factor of 3 is less than one in 200 million.

The relative time and space performance depends on p. Dr. Pugh suggests p = 0.25 for most cases. If the predictability of performance is important, then he suggests using p = 0.5 (the variability of the performance decreases with larger p).

Interestingly, the average number of references per node is only 1.33 when p = 0.25 is used. A BST has 2 references per node, so a skip list is more space-efficient.

Skip List Implementation

```
template <typename Comparable>
class SkipList {
private:
   class SkipListNode {
      public:
       void setDatum(const Comparable &datum);
       void setForward(int I, SkipListNode *f);
       void setSize(int sz);
       SkipListNode();
       SkipListNode(const Comparable& datum, int size);
       SkipListNode(const SkipListNode &c);
       ~SkipListNode();
       const Comparable & getDatum() const;
       int getSize() const;
       SkipListNode *getForward(int level);
```

Skip List Implementation (cont)

```
   private:   // SkipListNode
        int m_size;
        vector <SkipListNode*> m_forward;
        Comparable m_datum;
   };  // SkipListNode
public:
   SkipList()
   SkipList(int max_node_size, double probab);
   SkipList(const SkipList &);
   ~SkipList();
   int getHighNodeSize() const;
   int getMaxNodeSize() const;
   double getProbability() const;
   void insert (const Comparable &item);
   bool find(const Comparable &item);
   void remove(const Comparable &item);
```

Skip List Implementation (cont)

```
private: // SkipList
   SkipListNode *find(const Comparable *item,
                              SkipListNode *startnode);
   SkipListNode *getHeader() const;
   SkipListNode *findInsertPoint(const Comparable &item,
               int nodesize);
   void insert(const Comparable &item, int nodesize,
                     bool &success);

   int m_high_node_size;
   int m_max_node_size;
   double m_prob;
   SkipListNode *m_head;
};
```

find

```
Bool find(const Comparable &x)
{
   node = header node
   for(reference level of node from (nodesize-1) down to 0)
       while (the node referred to is less than x)
              node = node referred to
   if (node referred to has value x)
       return true
   else
       return false
}
```

findInsertPoint

Ordinary list insertion:

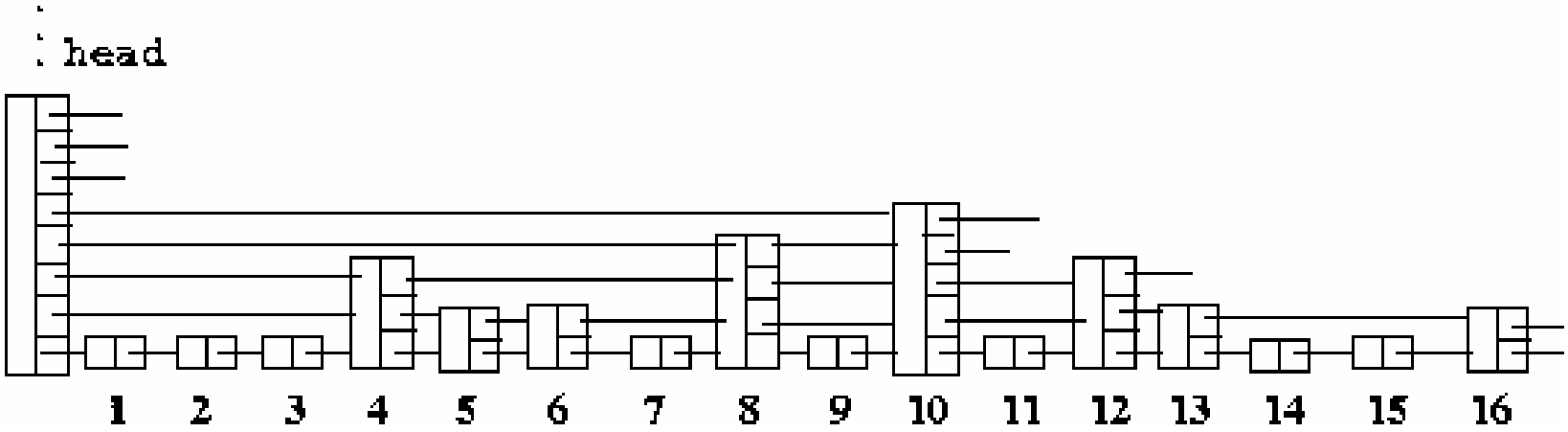
have handle (iterator) to node to insert in front of

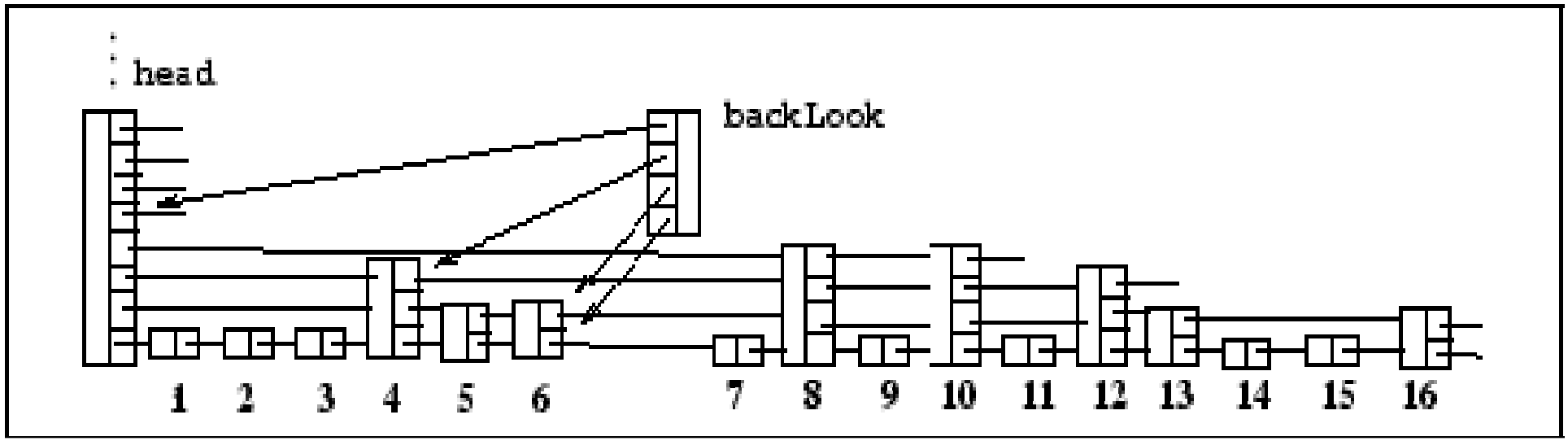
Skip list insertion:

need handle to all nodes that skip to node of given size at insertion point (all “see-able” nodes)

Use `backLook` structure with a pointer for each level of node to be inserted

Insert 6.5





In the figure, the insertion point is between nodes 6 and 7. "Looking" back towards the header, the nodes you can "see" at the various levels are

level	node seen
0	6
1	6
2	4
3	header

We construct a "backLook" node that has its forward pointers set to the relevant "see-able" nodes. This is the type of node returned by the `findInsertPoint` method

insert method

Once we have the backLook node returned by findInsertPoint and have constructed the new node to be inserted, the insertion is easy.

The public insert(const Comparable& x) decides on the new nodes size by random choice, then calls the overloaded private insert(const Comparable& x, int nodeSize) to do the work.

Code is available in Dr. Anastasio's HTML version of these notes