

EECS 498 System Design of a Search Engine

Winter 2026

Lecture 6: Hashing and serializing data

Nicole Hamilton

[https://web.eecs.umich.edu/~nham/
nham@umich.edu](https://web.eecs.umich.edu/~nham/nham@umich.edu)

agenda

1. Course details
2. Hashing
3. Serializing data

agenda

1. Course details.
2. Hashing
3. Serializing data

details

1. I'm halfway through reading your introductions. I'm really enjoying them.
2. I published the string/vector AG grades this morning. Several groups have extensions on the Utf8 assignment.
3. I was hoping to show off the group photos but we still have one team with an extension because they only found their 6th member last week.
4. Faces in a LOT of the photos are WAY too small to be recognizable and a lot of you have crazy hats photoshopped onto you. You have time to redo them if you want an extension.
5. I was also hoping to share survey results but I haven't gotten them back from my staff yet.
6. Questions about homework, project, anything else?

agenda

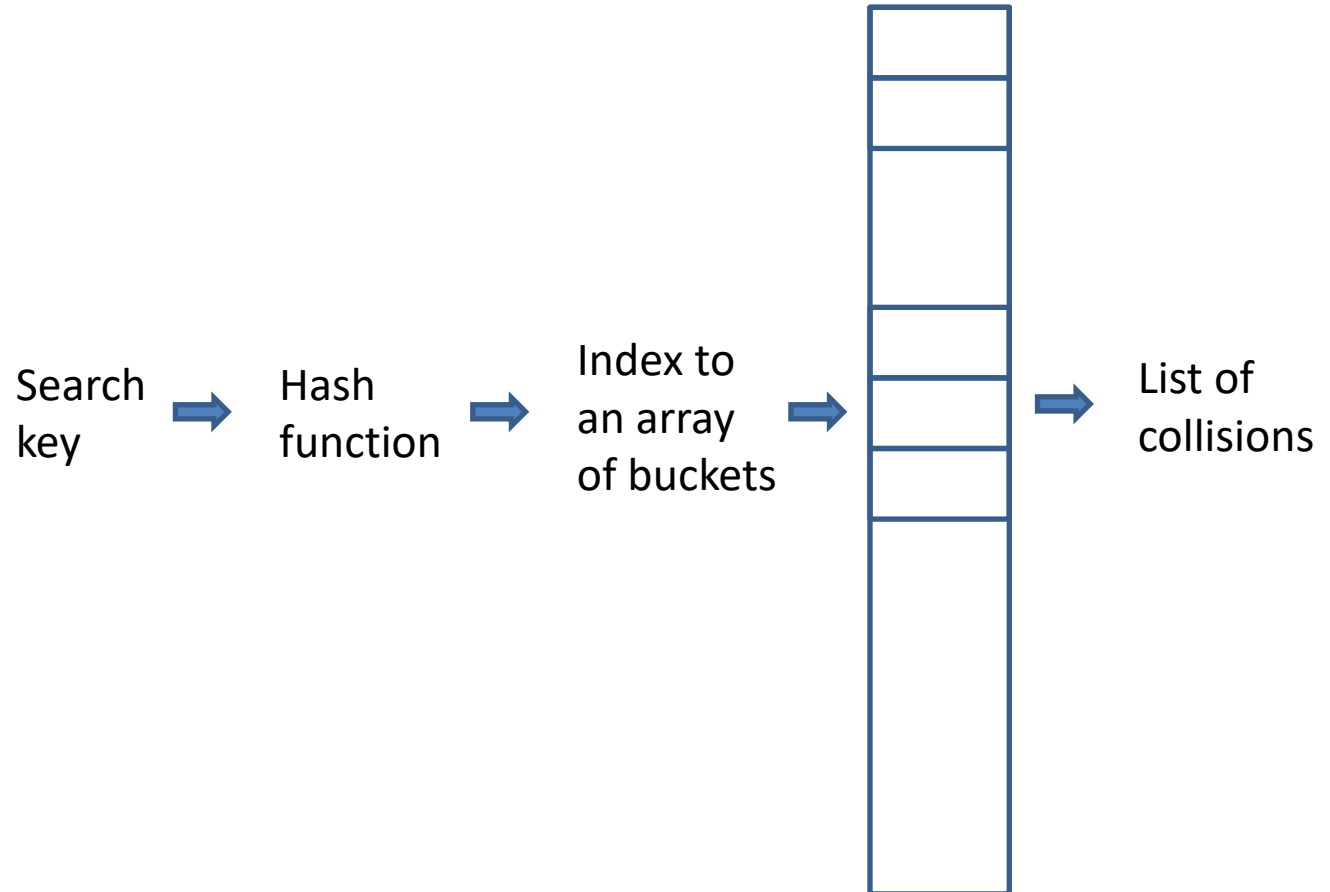
1. Course details
2. Hashing
3. Serializing data

hashing

Create an array of buckets to hold the items you wish to index based on a key field.

Create a simple function that can map the key to a bucket index.

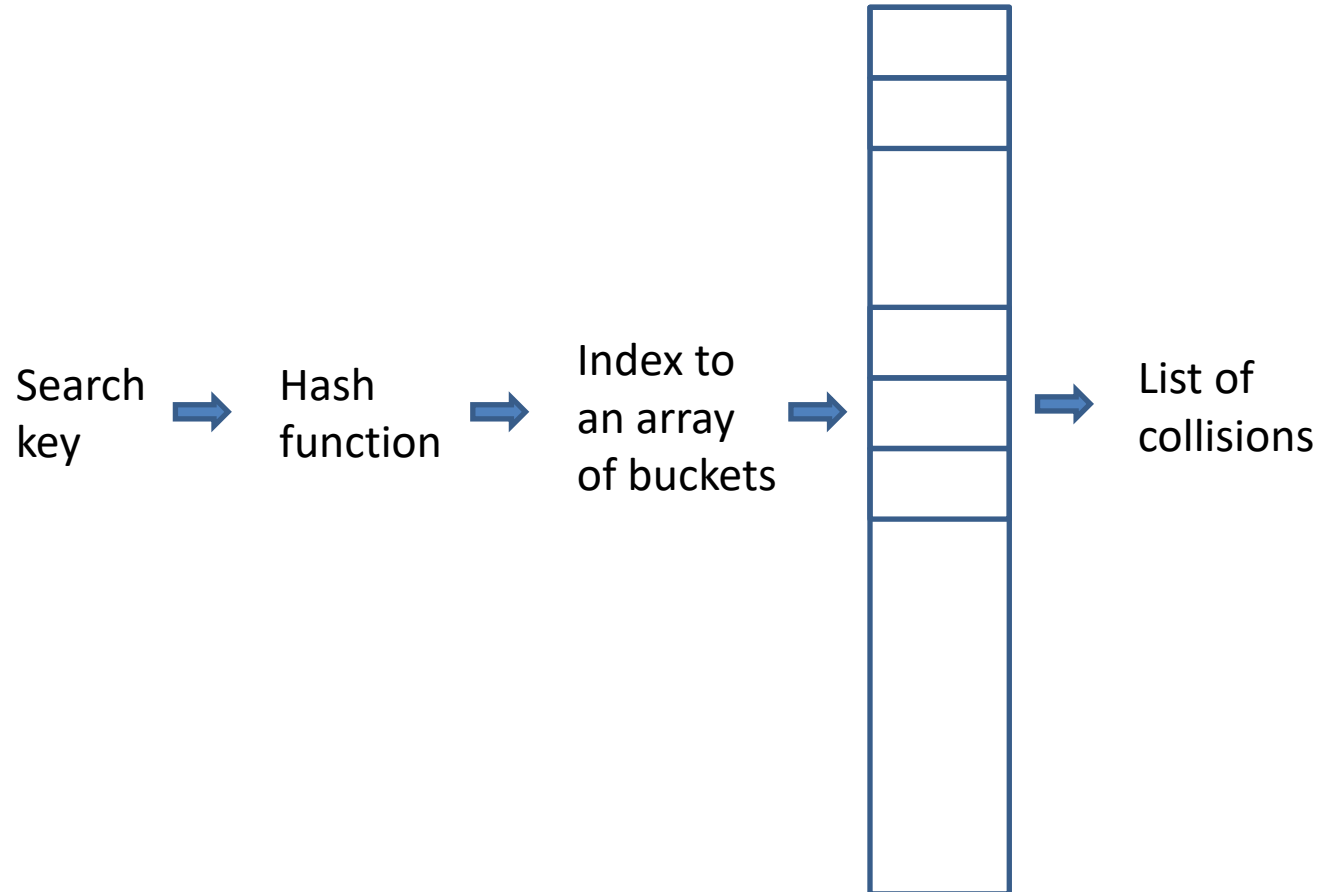
If multiple different keys map to the same bucket, that's called a collision.



hashing

Expensive to grow the table or deal with collisions, so we make the table pretty big.

No serious developer would use hashing except when they expect *lots* of entries that don't have to be kept in sort order, so there's no point in starting with only two buckets.

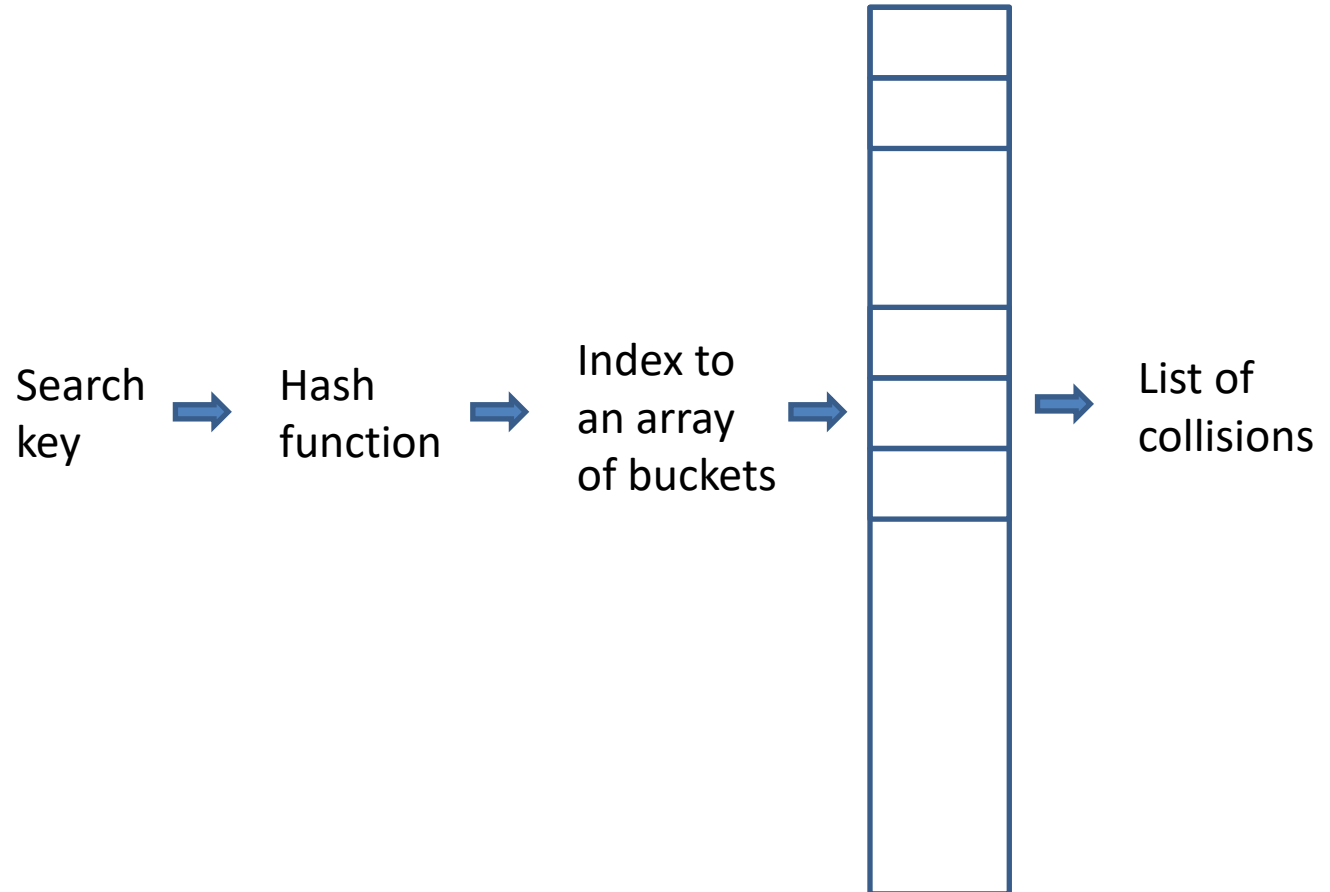


hashing

Start with 1024 or 4096 or some other big power-of-two number of buckets that you expect matches the problem.

Use the low bits of the hash function as an index.

You may/may not support growing the hash table.



hash function

Even a simple hash
can work pretty
well.

But what's the
problem with this
one?

```
unsigned Hash( const char *p )  
{  
    unsigned h = 0;  
    for ( ; *p; p++ )  
        h = ( h << 1 ) ^ *p;  
    return h;  
}
```

Fowler-Noll-Vo hash function

A more
complex
hash
using big
prime
numbers.

```
size_t fnvHash ( const char *data, size_t length )
{
    static const size_t FnvOffsetBasis =
        146959810393466560
    static const size_t FnvPrime =
        1099511628211ul;

    size_t hash = FnvOffsetBasis;
    for ( size_t i = 0; i < length; ++i )
    {
        hash *= FnvPrime;
        hash ^= data[ i ];
    }
    return hash;
}
```

polynomial CRCs

CRCs (cyclic redundancy codes) use shift registers.

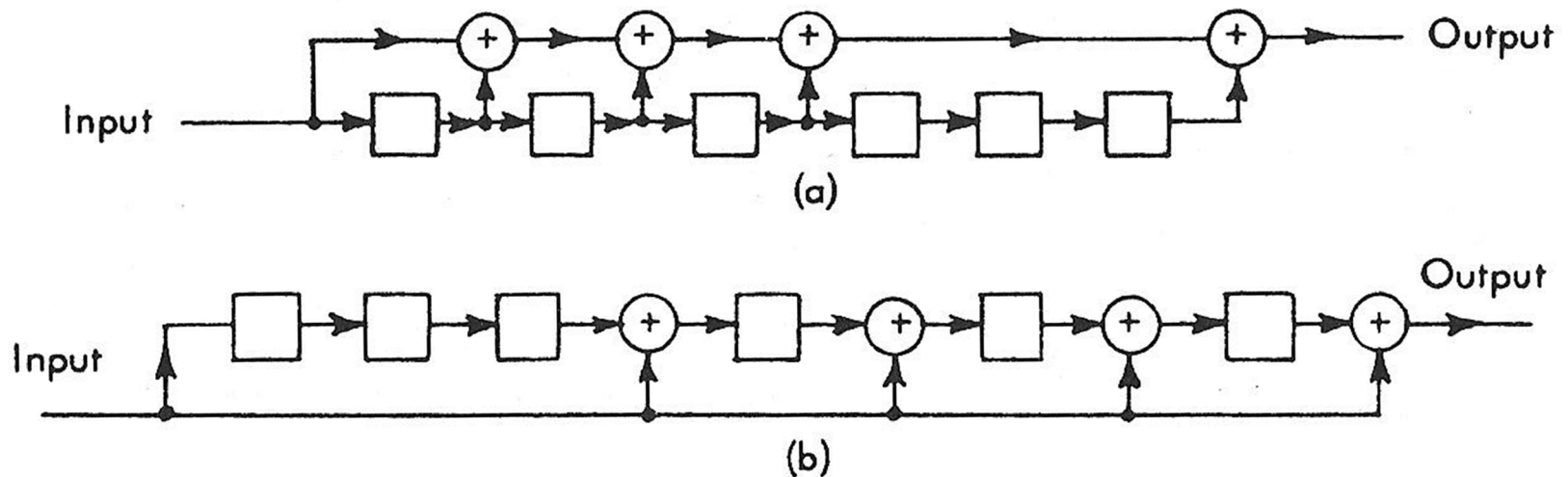


Figure 7.5. Circuits for multiplying by $X^6 + X^5 + X^4 + X^3 + 1$.

Source: W. Wesley Peterson and E.J. Weldon, Jr, *Error-Correcting Codes*, Second Edition, MIT Press, 1972, pp 178-179.

polynomial CRCs

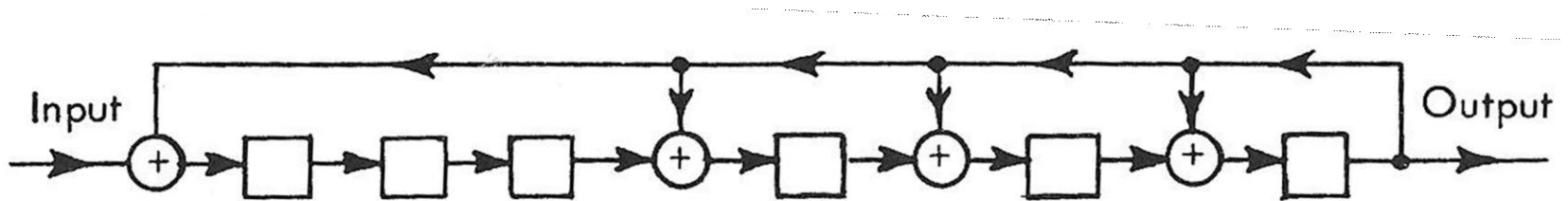


Figure 7.7. A circuit for dividing by $X^6 + X^5 + X^4 + X^3 + 1$.

```
/*  
Ethernet CRC  
Polynomial    x^32 + x^26 + x^23 + x^22 + x^16 + x^12 + x^11 + x^10 +  
               x^8 + x^7 + x^5 + x^4 + x^2 + x^1 + 1
```

Represent the polynomial as an unsigned number with each bit corresponding to a term of the polynomial set to one. E.g., the x^5 term means bit 5 (counting from the LSB as bit 0) is on. The x^{32} term is actually the feedback term and is "off the end" of the polynomial.

```
*/  
  
#define Ethernet_Polynomial    0x04c11db7  
#define Poly                   Ethernet_Polynomial
```

```
/* Calculate the CRC table entries.  crc(x) returns table entry x.  Use this
   routine to fill a table crctab[256] with the appropriate values.
*/
```

```
ulong crc( register ulong i )
{
    i <<= 24;

    i = (i << 1) ^ ((i & 0x80000000) ? Poly : 0);
    i = (i << 1) ^ ((i & 0x80000000) ? Poly : 0);
    i = (i << 1) ^ ((i & 0x80000000) ? Poly : 0);
    i = (i << 1) ^ ((i & 0x80000000) ? Poly : 0);
    i = (i << 1) ^ ((i & 0x80000000) ? Poly : 0);
    i = (i << 1) ^ ((i & 0x80000000) ? Poly : 0);
    i = (i << 1) ^ ((i & 0x80000000) ? Poly : 0);
    i = (i << 1) ^ ((i & 0x80000000) ? Poly : 0);

    return i;
}
```

```
static const unsigned long CRCTable[ ] =  
    { 0x00000000, 0x04c11db7, 0x09823b6e, 0x0d4326d9,  
      0x130476dc, 0x17c56b6b, 0x1a864db2, 0x1e475005,  
      0x2608edb8, 0x22c9f00f, 0x2f8ad6d6, 0x2b4bcb61,  
      0x350c9b64, 0x31cd86d3, 0x3c8ea00a, 0x384fbdbd,  
      0x4c11db70, 0x48d0c6c7, 0x4593e01e, 0x4152fda9,  
      0x5f15adac, 0x5bd4b01b, 0x569796c2, 0x52568b75,  
      0x6a1936c8, 0x6ed82b7f, 0x639b0da6, 0x675a1011,  
      . . .  
      0xbcb4666d, 0xb8757bda, 0xb5365d03, 0xb1f740b4    };
```

```
// To calculate a CRC, use the UpdateCRC( crc, c ) procedure to
// accumulate the data into the CRC. The crc argument is the
// accumulated CRC, c is the next character of data to be added.
// Initially, crc = 0.
```

```
unsigned long UpdateCRC( unsigned long crc, unsigned char c )
{
    return ( ( ( crc ) << 8 ) ^ CRCTable[ ( ( crc ) >> 24 ) ^ ( c ) ] );
}
```

buckets

Can either contain the object being indexed or a pointer to the object.

If it's the object itself, you need at least as many buckets as objects and each bucket will be larger.

If it's a pointer, it can be to a list.

collisions

You have to assume that you will have collisions.

When searching a hash bucket, you must compare the search keys to make sure that what you found is what you wanted.

If the key fields are long, you might decide to add a field to a bucket entry with the complete hash of the key and compare that before comparing the entire field.

collisions

Several ways to deal with collisions.

1. Increment the bucket number modulo the number of buckets and try there.
2. Use a second hash (or just different bits of the same hash) and try there.
3. Create a list off each bucket.
4. Try (1) or (2) and if not successful, create list.

collisions

When using a list for collisions, the insertion order matters.

The STL map template pushes collisions onto the front of the list.

Why did they do that and is that a good choice?

collisions

If query performance is more important than build time, collisions should be placed onto the end.

Why?

collisions

If query performance is more important than build time, collisions should be placed onto the end.

Why?

Because you expect to encounter the most frequent terms earlier in the build and you want those at the front.

collisions

If you use a list for collisions, most people normally think of something like a linked list.

Since you don't know what will have to go into the list, one way to deal with collisions by pushing new node onto the list (or onto the end).

collisions

But suppose you already know what the data looks like.

Could you use that information to build a better performant, smaller memory footprint, more efficient hash table to hold exactly that known data?

if you know the data

You might try several hash algorithms to see which gives you the lowest rate of collisions.

You could tune the number of buckets.

You could compress the collision lists. Instead of a linked list, it could be an array or a serialized stream.

compressing the list

Buckets contain pointers or, even better, offsets into one large contiguous buffer containing the items stored in the hash table.

Each list is a contiguous series of bytes representing a concatenation of the entries on a collision list.

If you only need to read them serially, no problem if the entries vary in size, as long as you can tell them apart.

You do need to know when you've reached the end, typically with a sentinel.

agenda

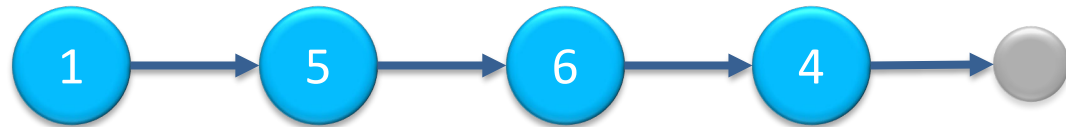
1. Course details
2. Hashing
3. Serializing data

serializing data structures

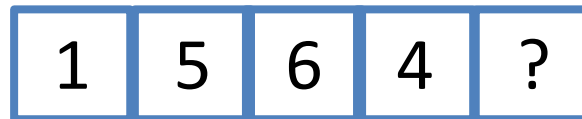
Common problem in CS is that you have a complex data structure that serialize, e.g., to store in a file or send over a wire.

lists are easy

```
// List
struct node
{
    int  datum;
    node *next;
};
```



Can be transformed into contiguous bytes with terminator (aka sentinel) to mark the end.



But it does require a way of marking the end with a value that can't be part of the list, as we do with the null character in C-strings.

envp

envp is an example.

Environment variables are passed as an array of pointers. The end is a null pointer.

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

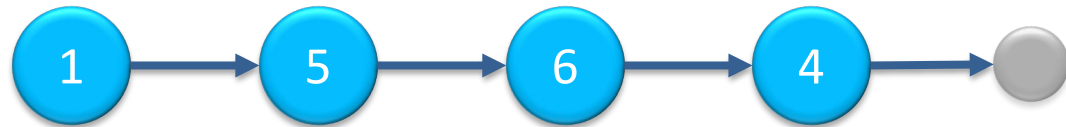
int main( int argc, char **argv,
          char **envp )
{
    for ( int i = 0; envp[ i ]; i++ )
        cout << envp[ i ] << endl;
}
```

```
tcsh-1% cat envp.cpp
#include <iostream>
using namespace std;

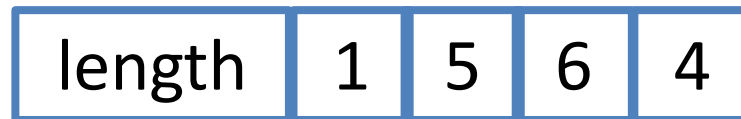
int main( int argc, char **argv,
          char **envp )
{
    for ( int i = 0; envp[ i ]; i++ )
        cout << envp[ i ] << endl;
}
tcsh-2% g++ envp.cpp -o envp
tcsh-3% ./envp
HOSTTYPE=x86_64-linux
LANG=C.UTF-8
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin: ...
:
HOME=/mnt/c/Users/hamil
USER=nicole
LOGNAME=nicole
SHELL=/bin/tcsh
WSL_DISTRO_NAME=Ubuntu
:
```

lists are easy

```
// List  
struct node  
{  
  int  datum;  
  node *next;  
};
```



A second way would be to put a length code at the front.

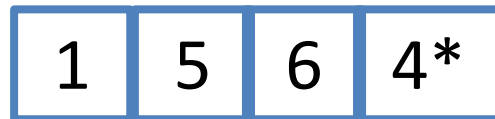


lists are easy

```
// List
struct node
{
    int  datum;
    node *next;
};
```



A third way may be to put something in the last element that marks it as the end.

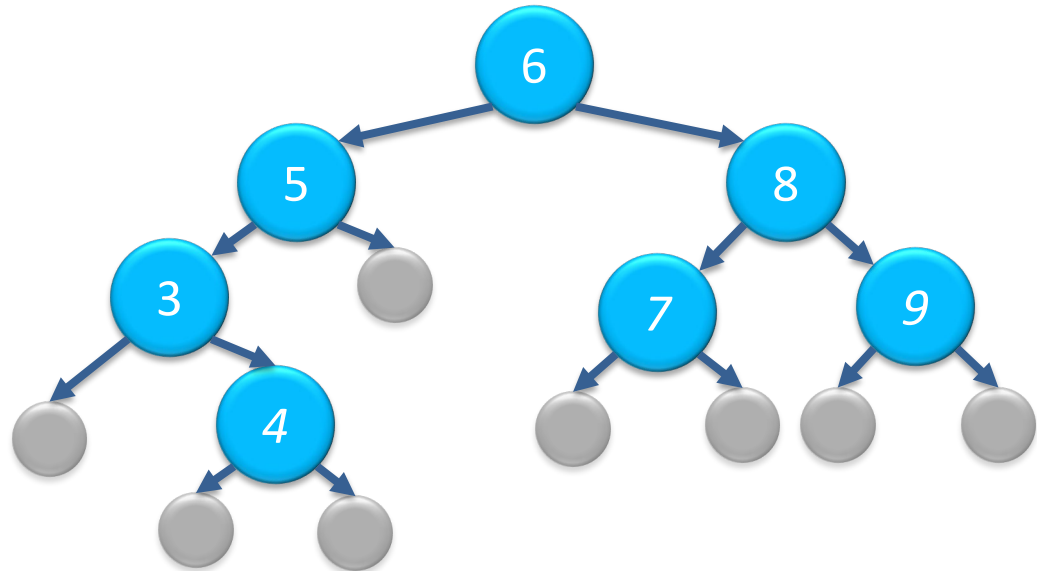


For example, if entries are variable length and need a length code used only when linear searching the list to go from one element to the next, the final entry could have an invalid length = 0 to mark the end.

binary trees are harder

Insertion order matters, so traversal order when serializing also matters.

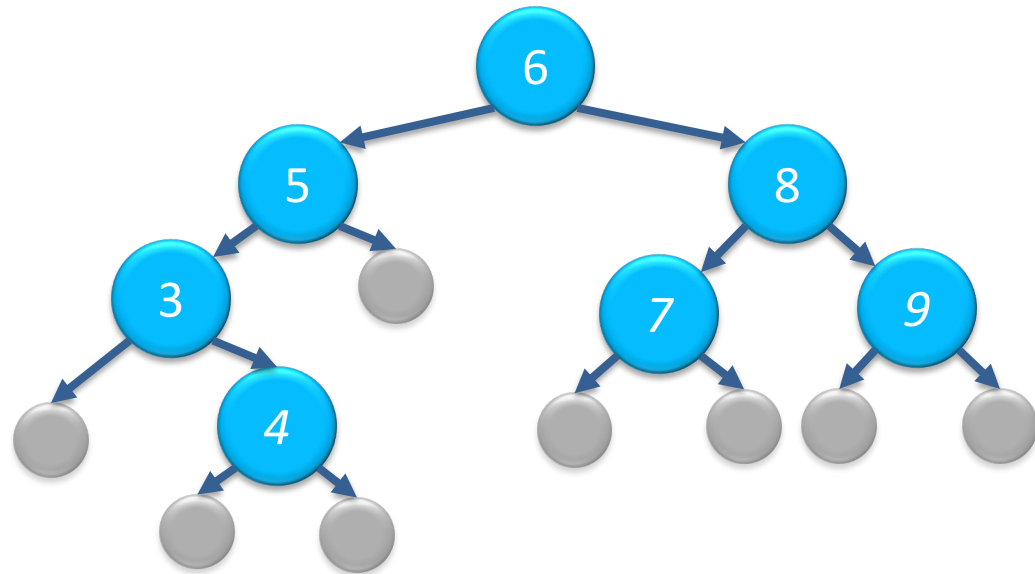
```
// Binary tree
struct node
{
    int  datum;
    node *left,
        *right;
};
```



insertion order

The shape of the tree and how fast it can be searched depends on the insertion order. The root is always the first thing that was inserted.

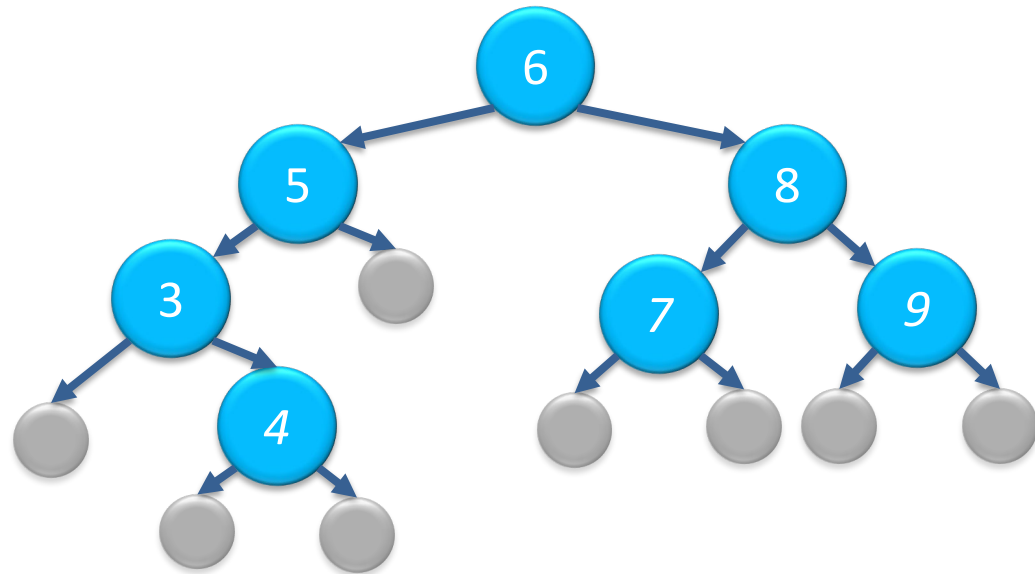
An insertion order 6, 5, 8, 3, 7, 9, 4 (and many others) would give us this relatively balanced tree.



insertion order

$O(\log n)$ performance assumes the tree will be balanced, so every comparison eliminates half the remaining tree.

We also hope the most frequently accessed nodes will be at the top.

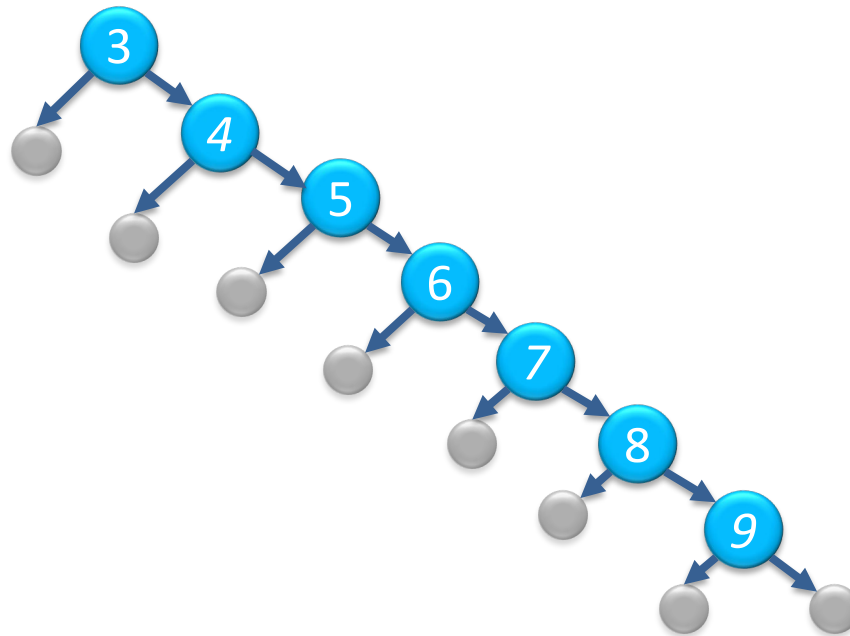


insertion order

But we could be unlucky.

If the insertion order was 3, 4, 5, 6, 7, 8, 9, we would get this, called a stick.

Perf drops to $O(n)$, not $O(\log n)$.

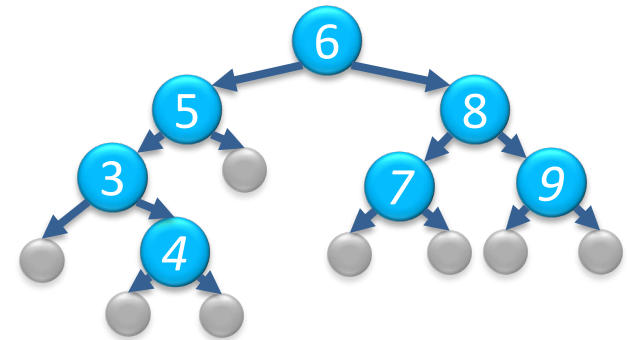


insertion order

If we used a binary tree to build a dictionary of words and counts as we crawl the English web:

1. Will it look more like the top or the bottom?
2. Will it do better or worse than $\log_2(n)$ comparisons on average?

Why or why not?

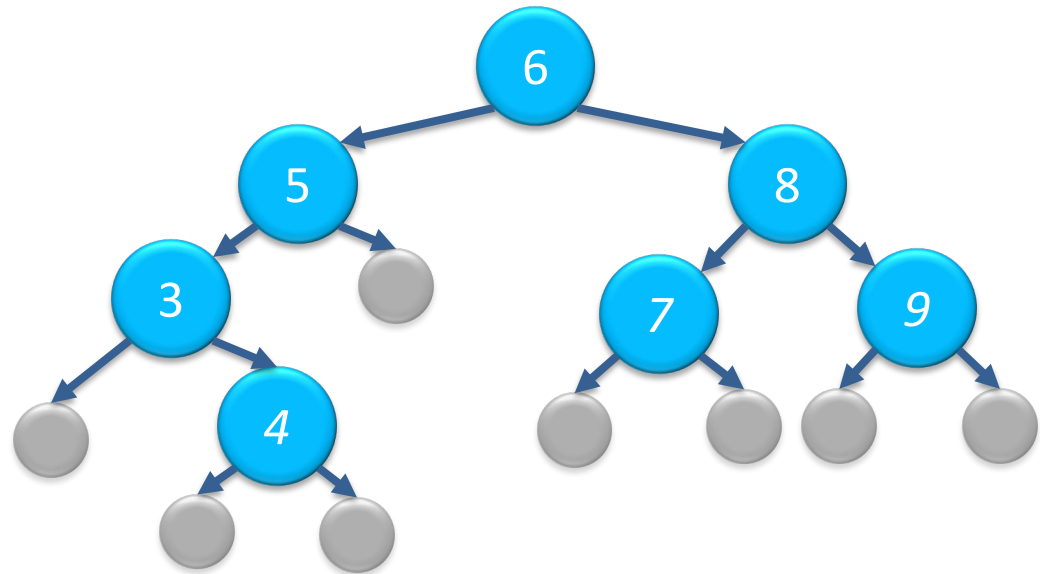


traversing a tree

Traversing a tree, visiting all the nodes, we need to visit the root and everything in the left and right subtrees.

Conventionally, we always traverse left before right.

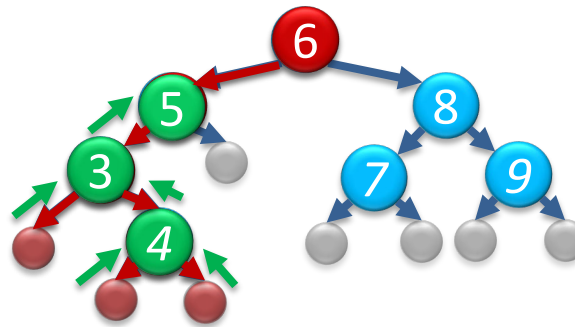
But we can choose when to visit the root.



inorder traversal

Print the left subtree, then the root, then the right.

```
void inorder( node *n )
{
    if ( n )
    {
        inorder( n->left );
        cout << n->datum << endl;
        inorder( n->right );
    }
}
```



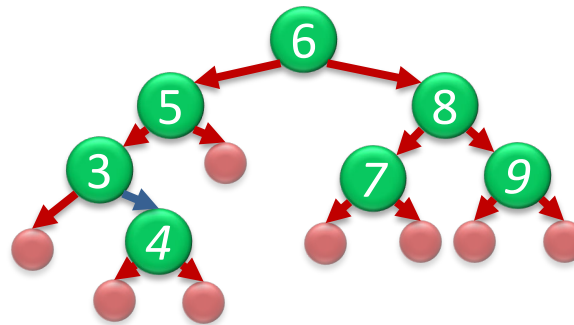
Prints

3
4
5

inorder traversal

Print the left subtree, then the root, then the right.

```
void inorder( node *n )
{
    if ( n )
    {
        inorder( n->left );
        cout << n->datum << endl;
        inorder( n->right );
    }
}
```



Prints

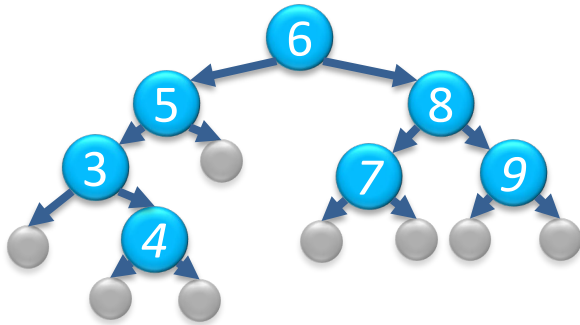
3
4
5
6
7
8
9

If we keep going, all the nodes are printed in sorted order because anything that's less is printed before the root and anything that's greater is printed after.

inorder traversal

If we inorder traverse a tree, then feed that into a new tree, we get a stick.

Original tree



Inorder traversal

3
4
5
6
7
8
9

New tree



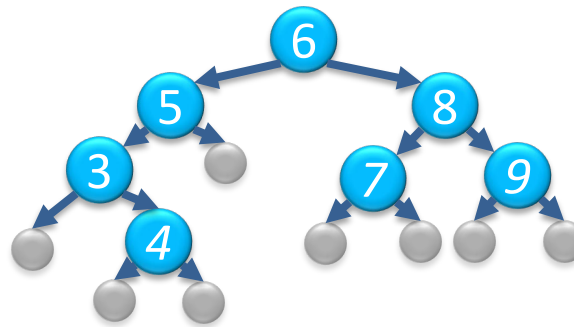
How could we copy a tree so we get the same shape?

We need to do the root first.

preorder traversal

Print the root first.

```
void preorder( node *n )
{
    if ( n )
    {
        cout << n->datum << endl;
        preorder( n->left );
        preorder( n->right );
    }
}
```



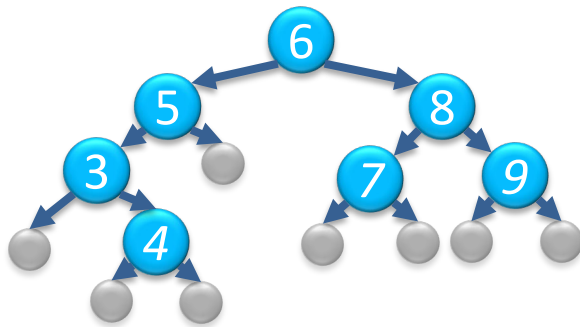
Prints

6
5
3
4
8
7
9

preorder traversal

If we preorder traverse a tree, then feed that into a new tree, we get the same tree.

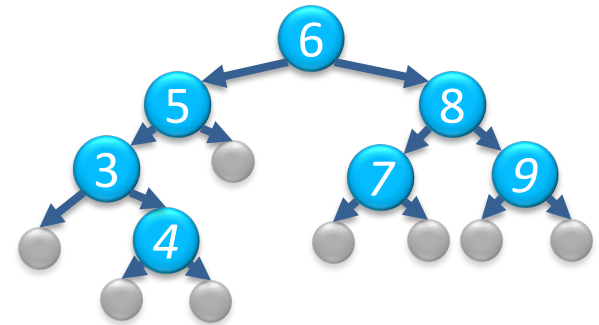
Original tree



Preorder traversal

6
5
3
4
8
7
9

New tree

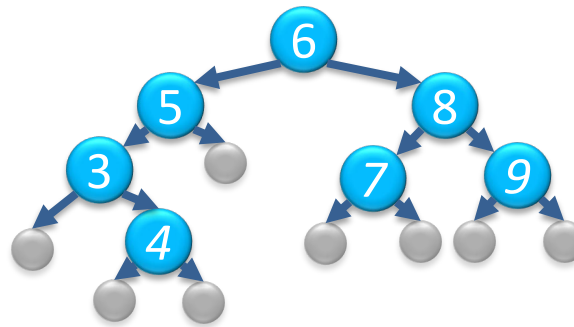


Useful for copying in memory or for serializing a tree, e.g., across a network.

postorder traversal

Print the root last.

```
void postorder( node *n )
{
    if ( n )
    {
        postorder( n->left );
        postorder( n->right );
        cout << n->datum << endl;
    }
}
```

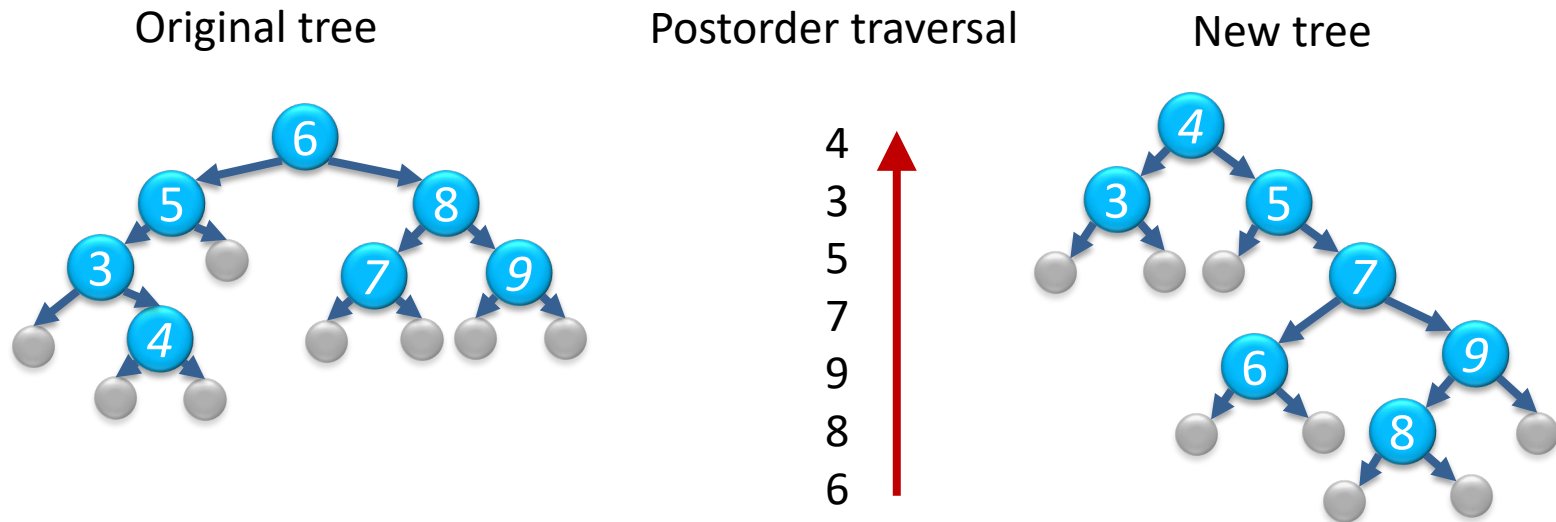


Prints

4
3
5
7
9
8
6

postorder traversal

If we postorder traverse a tree, then feed that into a new tree, we get a new random tree.

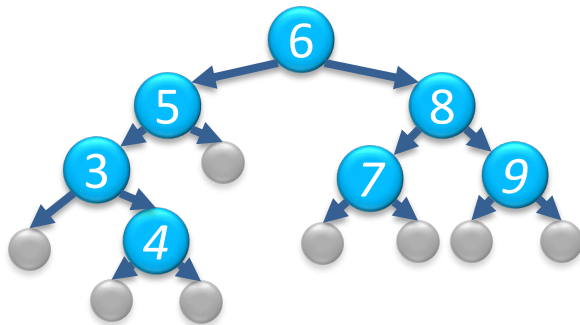


But if we feed the list in reverse into a new tree, we get the original.
Is postorder the reverse of preorder?

postorder vs preorder

No, postorder is not the reverse of preorder. But why?

Original tree



Postorder traversal

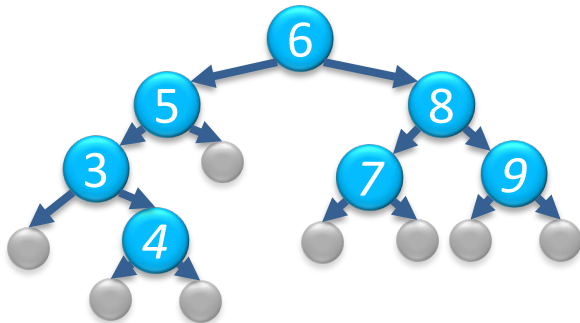
4
3
5
7
9
8
6

Preorder traversal

6
5
3
4
8
7
9

postorder vs preorder

Postorder and preorder are not the reverse of each other because they aren't completely reversed. Left is always done before right.



Preorder	Postorder
6	4
5	3
3	5
4	7
8	9
7	8
9	6

```
void preorder( node *n )
{
    if ( n )
    {
        cout << n->datum << endl;
        preorder( n->left );
        preorder( n->right );
    }
}

void postorder( node *n )
{
    if ( n )
    {
        postorder( n->left );
        postorder( n->right );
        cout << n->datum << endl;
    }
}
```

three traversal methods

```
void inorder( node *n )
{
    if ( n )
    {
        inorder( n->left );
        cout << n->datum << endl;
        inorder( n->right );
    }
}
```

```
void preorder( node *n )
{
    if ( n )
    {
        cout << n->datum << endl;
        preorder( n->left );
        preorder( n->right );
    }
}
```

```
void postorder( node *n )
{
    if ( n )
    {
        postorder( n->left );
        postorder( n->right );
        cout << n->datum << endl;
    }
}
```

They differ only in when the root is printed.

Inorder is convenient for listing a dictionary in sorted order.

Preorder is useful for copying a tree.

When would you use postorder?