Hashing and Sequence Assembly

Pre/Suffixes are nodes; reads are edges

- Each node will stand for either a prefix or a suffix.
  - Prefixes and suffixes are the same length, so each node will be used to represent both.

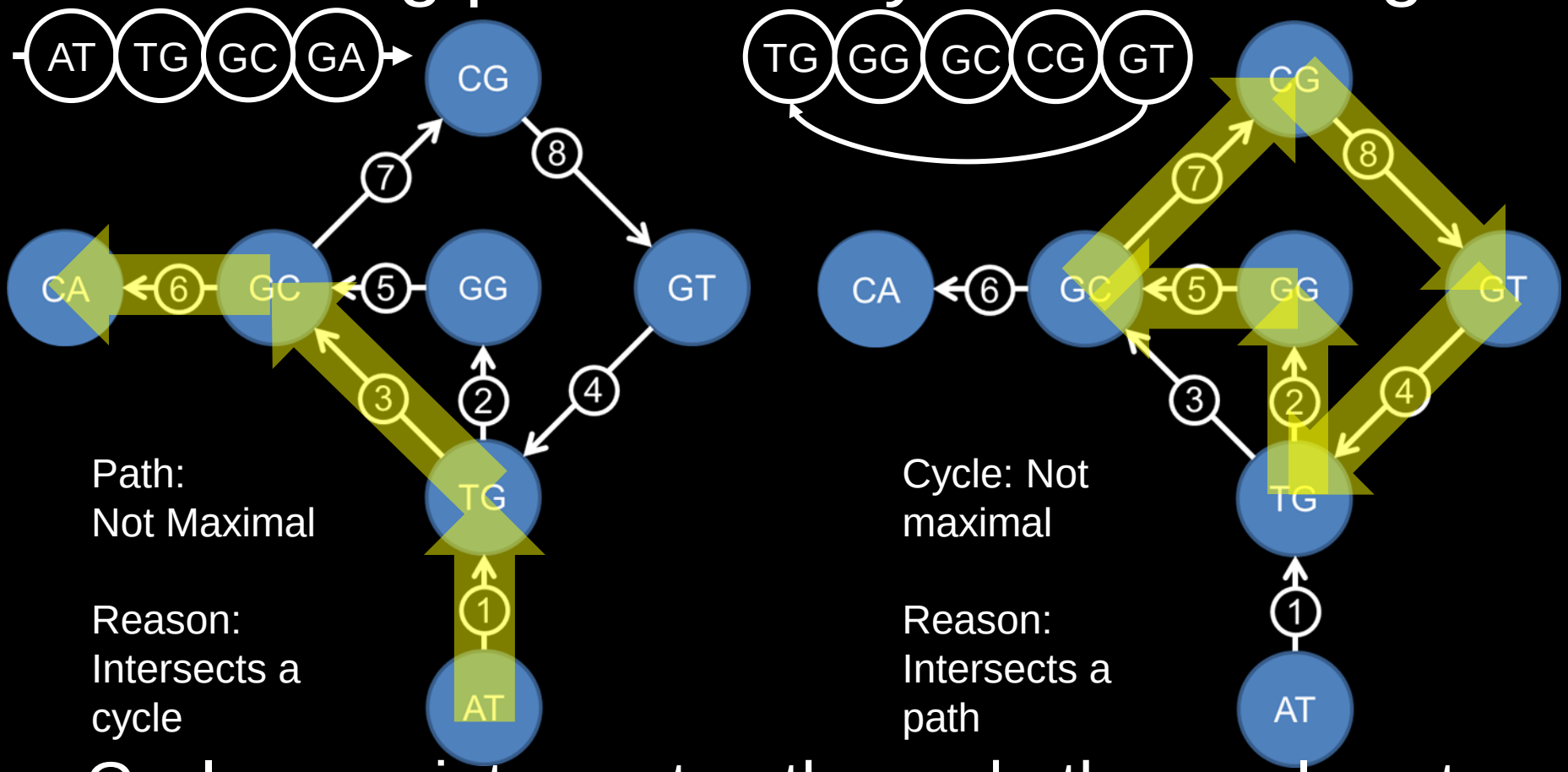
- Each edge will stand for a read, which connects a prefix to a suffix

Paths connect reads into longer sequences

- Paths indicate which reads connect and how
  - based on prefixes and suffixes
- Prefixes and Suffixes always the same length
- Contigs are maximal (longest possible) paths

- How do you know a contig is maximal?
  - It does not intersect any unused cycles
  - Its end nodes cannot be extended over any unused edges (backwards or forwards)

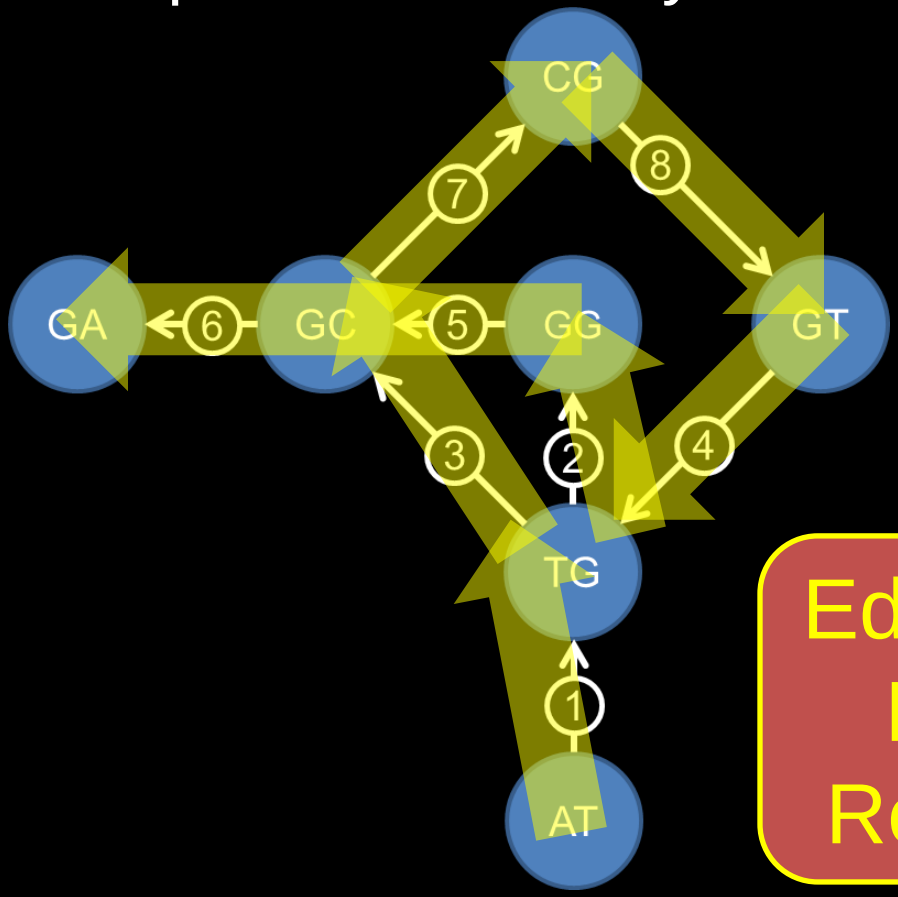
Assembling paths and cycles for contigs



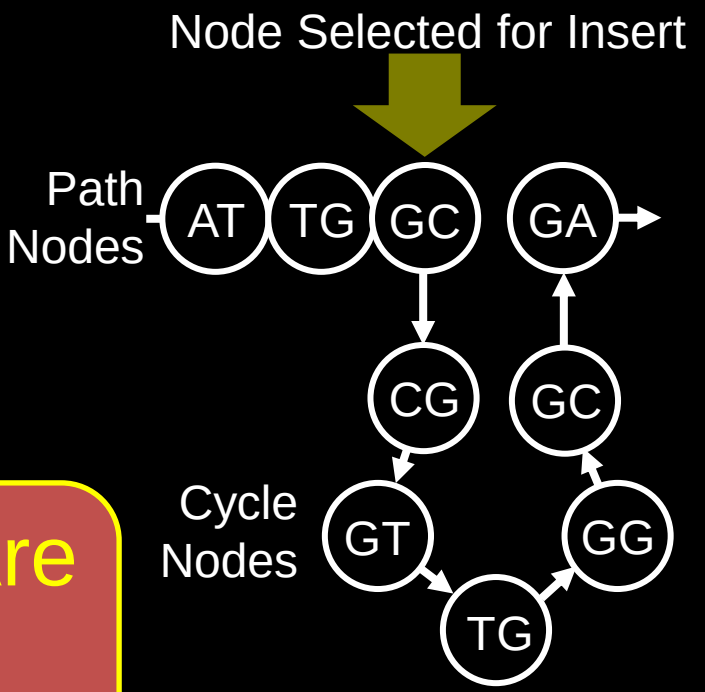
- Cycles can intersect paths and other cycles at nodes, but they CANNOT share edges.
- As you generate paths and cycles, mark edges used, so that they don't accidentally get reused

Assembling paths and cycles for contigs

- Inserting cycles into paths and other cycles happens at exactly one node
  - Even if the cycle intersects the path at multiple nodes, pick a node. Any node.



Edges Are Never Re-used!



Cycles create ambiguity in final assemblies

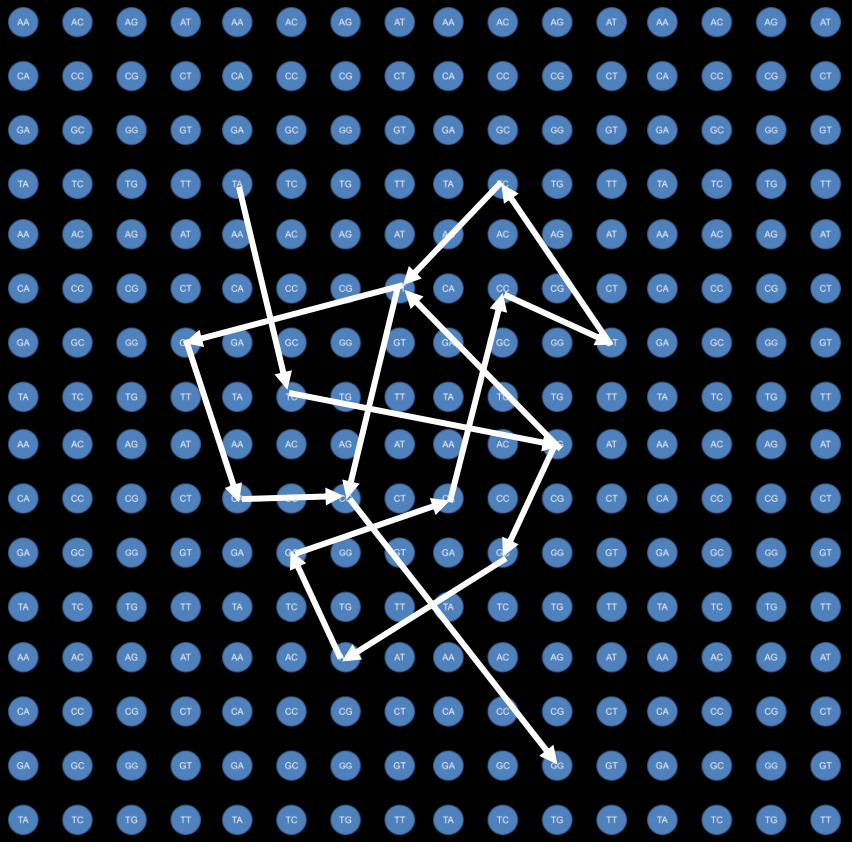
- Cycles can intersect a path or other cycles at multiple points
- One point must be chosen for insertion
- The fact that there is a choice means that there are multiple possibilities for the final outcome.
- We will talk about ways to fully deal with this ambiguity later

Representing large graphs in memory

- The exponential space of nodes:
  - 1,073,741,824
  - In larger assembly efforts (e.g. mammalian genomes, etc) this can be 4^{31}
- The space of edges is the number of reads:
  - 60,000
- It is hard to represent nodes in memory.
- Fortunately, you don't have to.

Lets look at the space of nodes

- There are many more nodes than edges.
- This is because we are numbering nodes that we will never use:
- Lets number only the nodes that we use



How do we number only nodes we use?

- First, we need a nonredundant index for every node
- Remember first that a node is a prefix or suffix with 15 nucleotides in it:

$$n_0\, n_1\, n_2\, n_3\, n_4\, n_5\, n_6\, n_7\, n_8\, n_9\, n_{10}\, n_{11}\, n_{12}\, n_{13}\, n_{14}$$

- Where n_0 is either A,C,G, or T
- Let A=0, C=1, G=2, T=3, and the expression:

$$V = n_0(4^0) + n_1(4^1) + \ldots + n_{13}(4^{13}) + n_{14}(4^{14})$$

- Stand for the whole prefix/suffix

- This ensures that one integer stands for every possible node. We'll call this a node index.

Some possible nodes and their indices

Example 1:

Read	Index
AAAAAAAAAAAAAAA	0

$V = n_0(4^0) + n_1(4^1) + \ldots + n_{13}(4^{13}) + n_{14}(4^{14})$

$V = 0(4^0) + 0(4^1) + \ldots + 0(4^{13}) + 0(4^{14}) = 0$

Example 2:

Read	Index
ACAAAAAAAAAAAAA	4

$V = n_0(4^0) + n_1(4^1) + \ldots + n_{13}(4^{13}) + n_{14}(4^{14})$

$V = 0(4^0) + 1(4^1) + \ldots + 0(4^{13}) + 0(4^{14}) = 4$

Example 3:

Read	Index
GCAAAAAAAAAAAAA	6

$V = n_0(4^0) + n_1(4^1) + \ldots + n_{13}(4^{13}) + n_{14}(4^{14})$

$V = 2(4^0) + 1(4^1) + \ldots + 0(4^{13}) + 0(4^{14}) = 4$

Example 4:

Read	Index
AAAAAAAAAAAAAAG	536,870,912

$V = n_0(4^0) + n_1(4^1) + \ldots + n_{13}(4^{13}) + n_{14}(4^{14})$

$V = 0(4^0) + 0(4^1) + \ldots + 0(4^{13}) + 2(4^{14}) = 536,870,912$

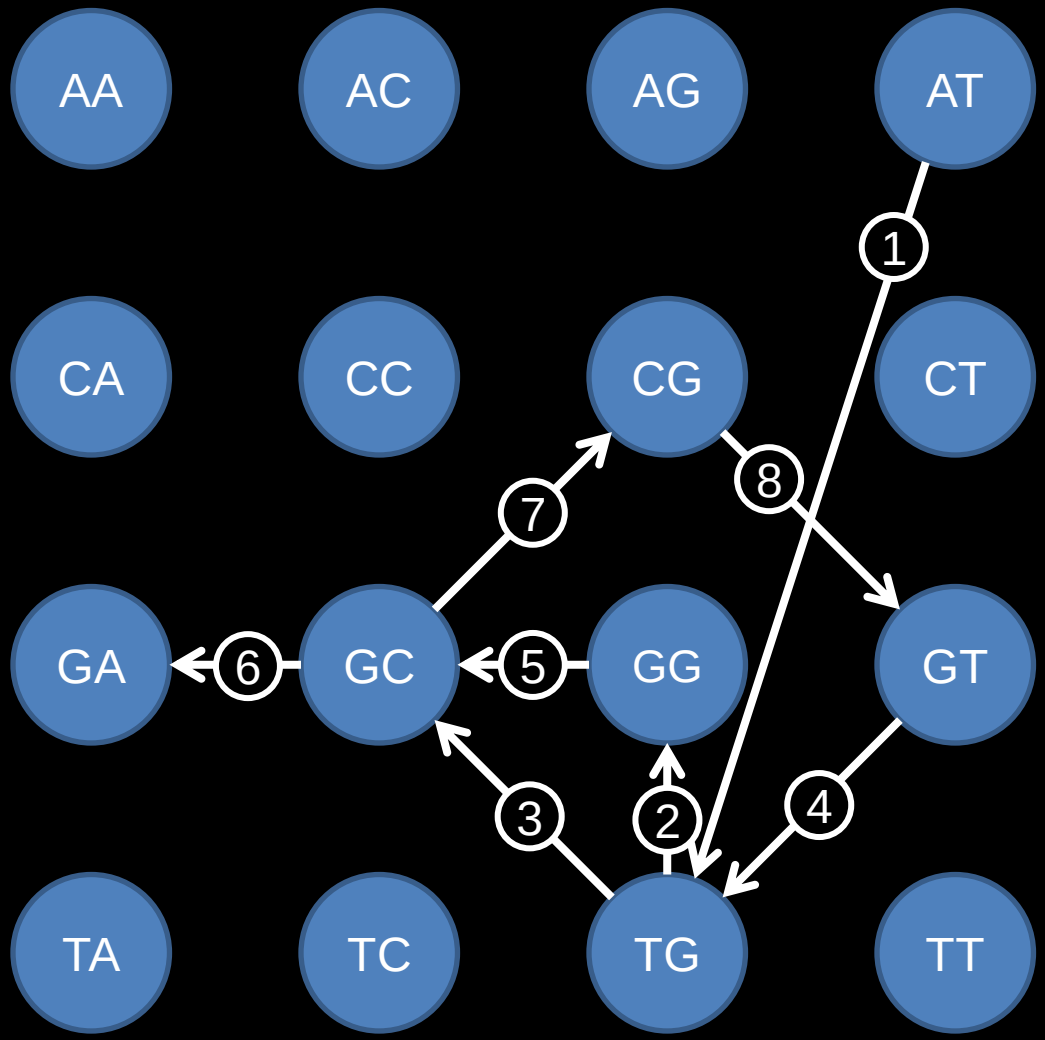
Hashing the indices to save space

- Suppose we just wanted to make a (nonredundant) list of the nodes that were used in each of our reads
- We could just make a list of the indices
- But we don't care about the indices of nodes that are not touched by any read

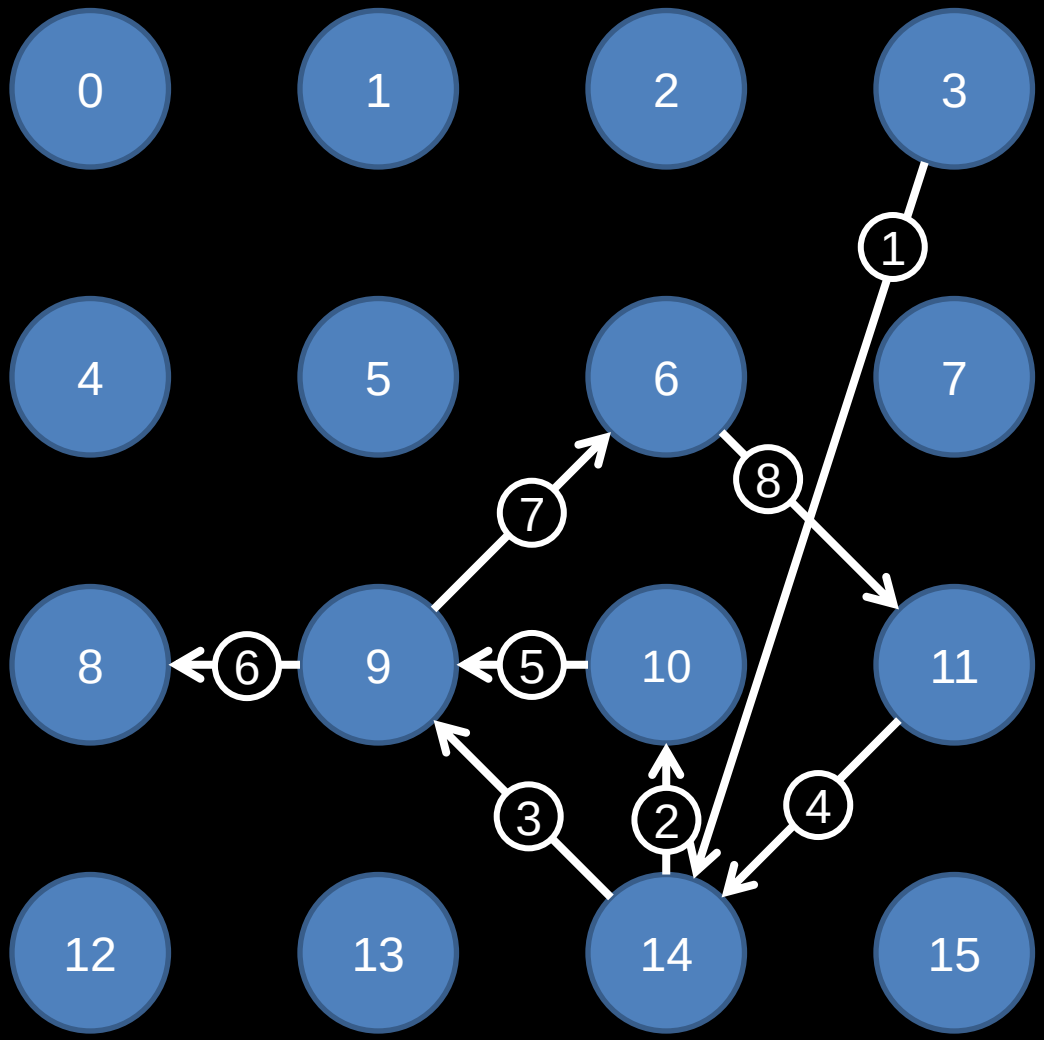
- So why not just use their position on the list, instead of the index?

#	Node Index
0	0
1	4
2	6
3	46
4	37,984,105
5	2,234,001
6	536,870,912
7	400,002,182
8	80,201,117
9	58

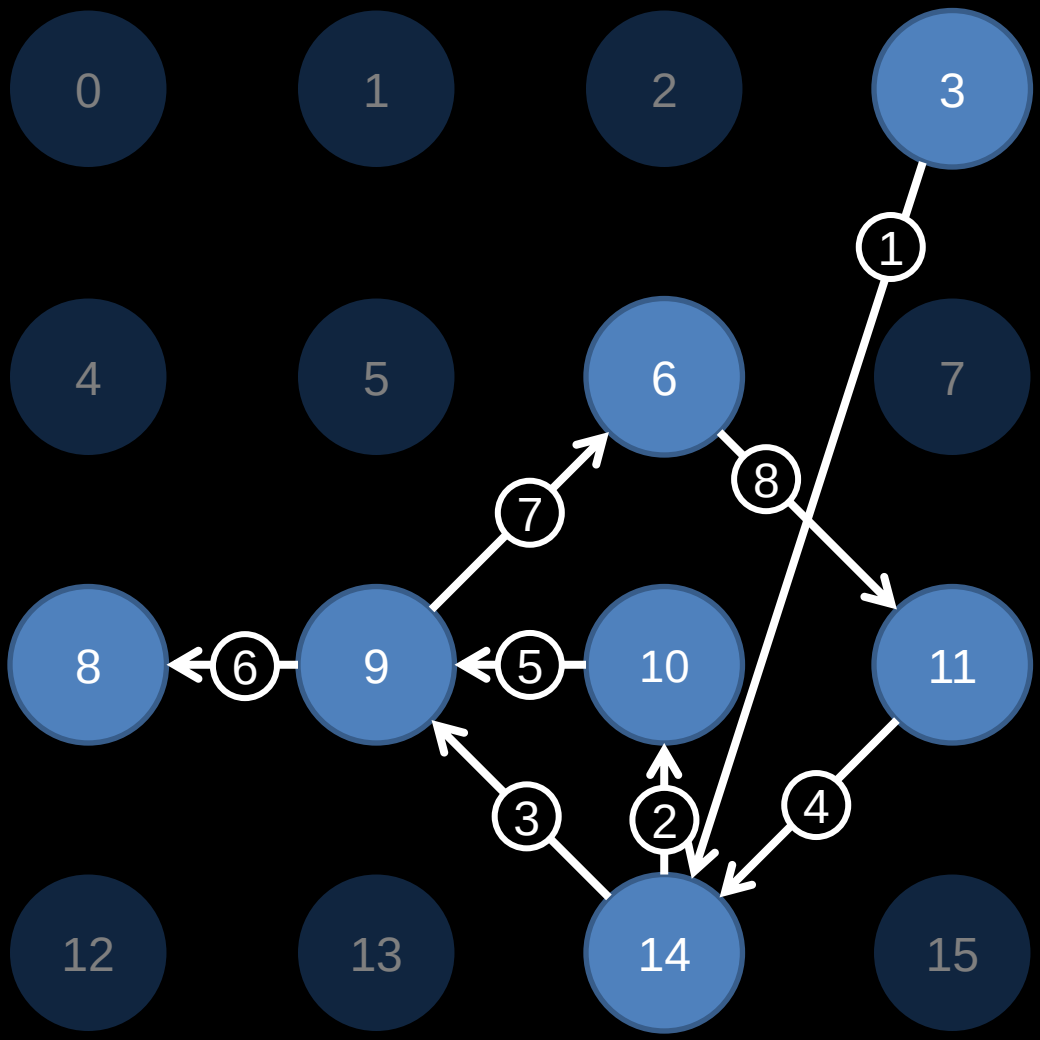
The graph we built before



The graph, substituted with Node Indices

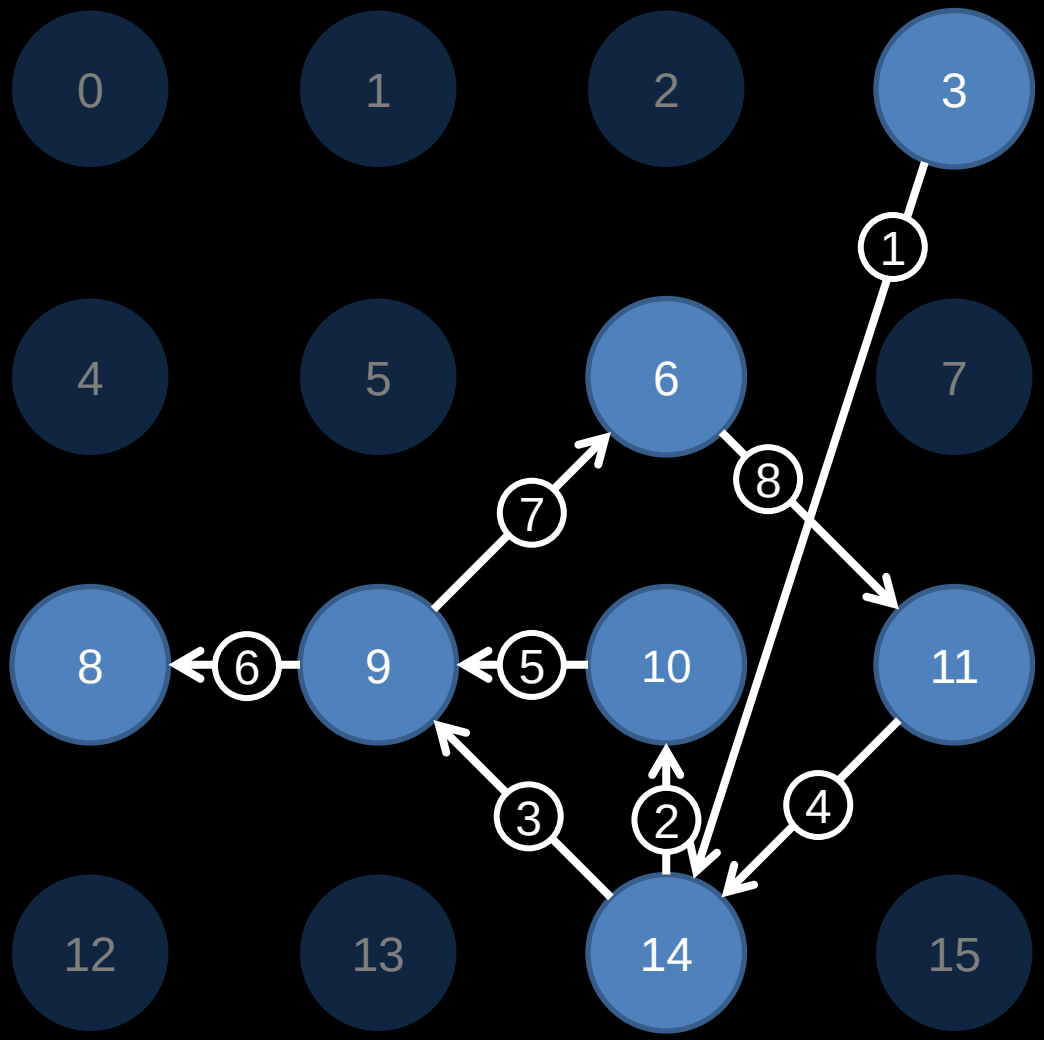


Grey out the nodes we don't use



Lets store the nodes we do use on a list

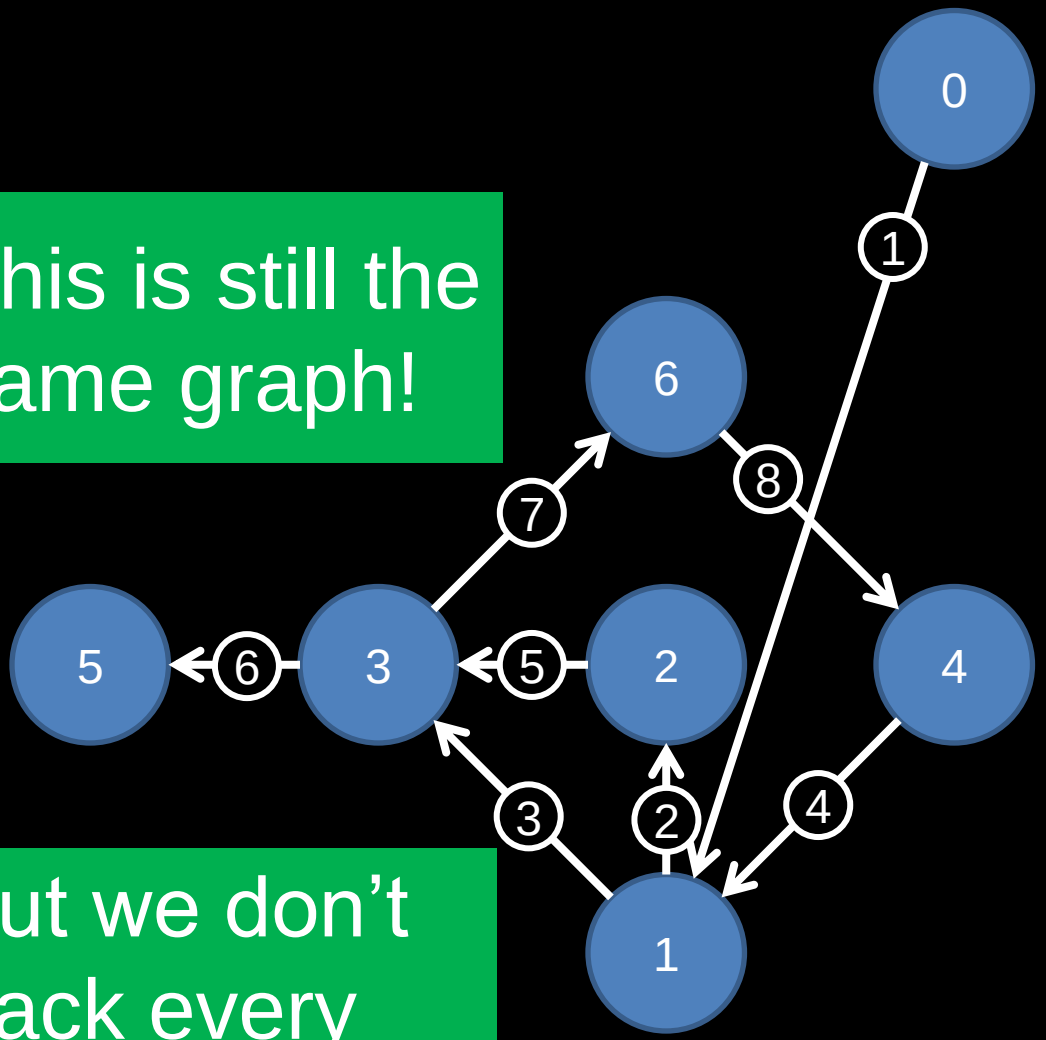
#	Node Index
0	3
1	14
2	10
3	9
4	11
5	8
6	6



If we renumbered the nodes with the list

#	Node Index
0	3
1	14
2	10
3	9
4	11
5	8
6	6

This is still the same graph!



But we don't track every node index

This isn’t the whole story yet

- What we’ve seen so far:
  - We can renumber the nodes in the graph that we use so that we don’t have to track all the nodes in the graph
- What we haven’t seen so far:
  - How do we store the graph structure in a useful way without having to remember lots of things?
- To do this, we need to know what a hash table does

Hash Table

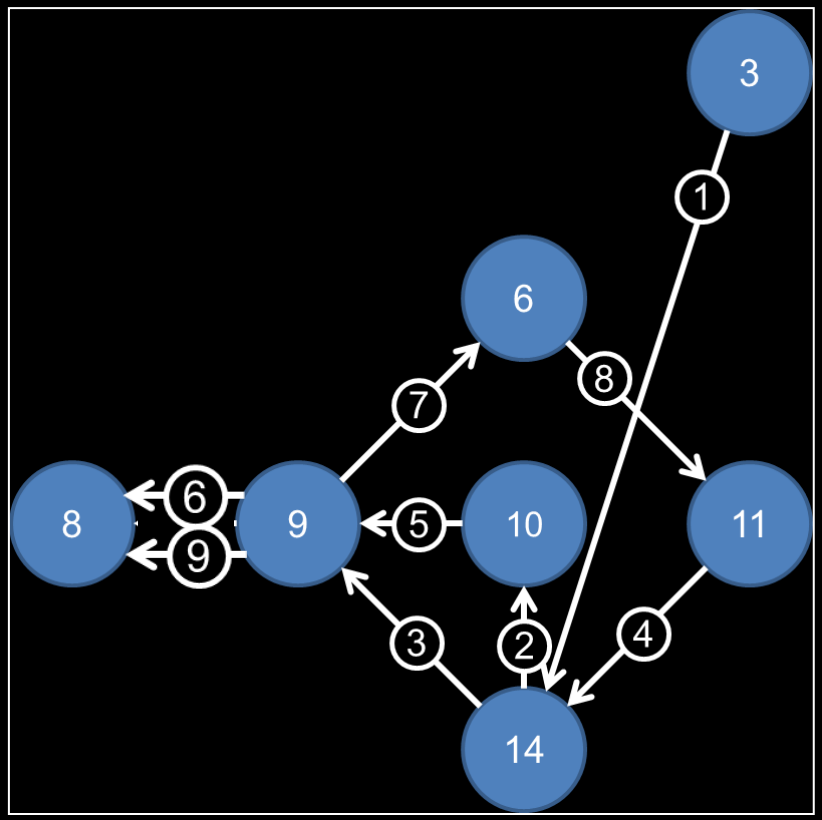
- A hash table is a list that holds “hash keys” in separate buckets. In this case, one key per bucket.
- You can add and remove things from the table
- It can tell you if it is holding a particular key without looking through the whole list

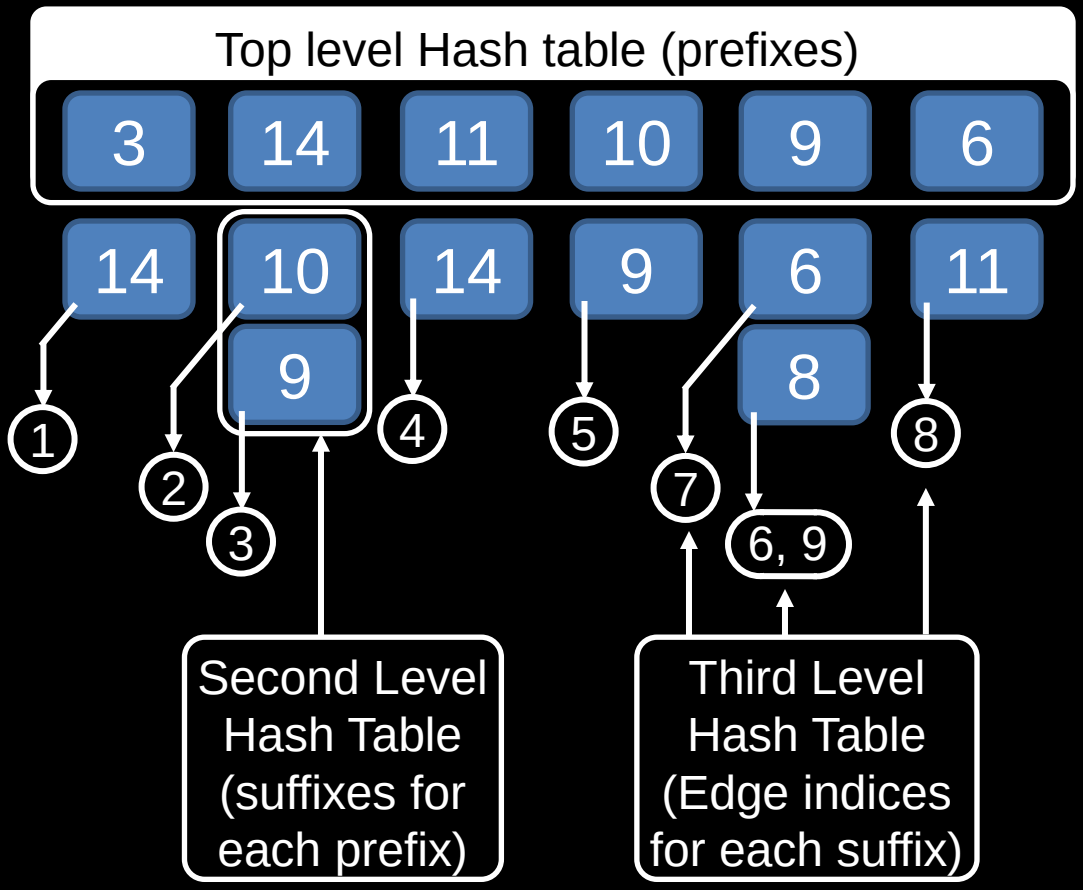
bucket	Hash Key
0	3
1	14
2	10
3	9
4	11
5	8
6	6

- If I wanted to store the node indices in a hash table, each node index would be a hash key.

Using the hash table to store edges

- An edge is two nodes:
  - the prefix node and the suffix node
  - You need nested hash tables





A hash table you can use for this purpose

/proj/cse308/Project1/setDemo

set_t

- A Dynamic Hash Table implemented in C

```
set_t mySet = alloc_set(0);
set_t mySet = alloc_set(SP_MAP);
```

- Sets are allocated in the C style
  - "0" to hold integers
  - "SP_MAP to hold pointers (Objects, other sets)
- A demonstration is implemented in setDemo

```
cd /proj/cse308/Project1/setDemo;
./example -set
```

Build the Edge Table with set_t

- First, instantiate the set. Lets call it “outerSet”

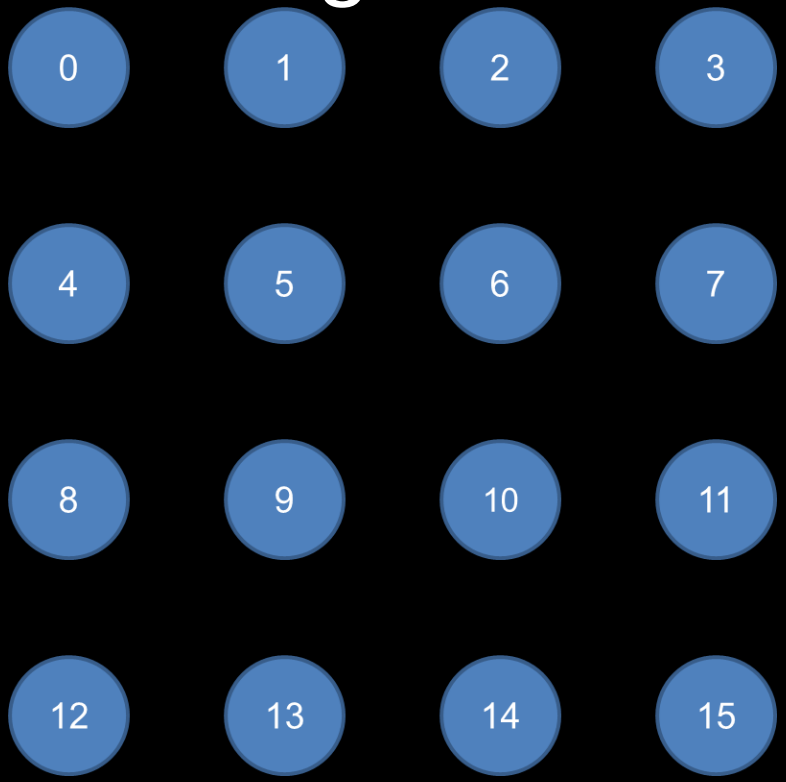
```
set_t outerSet = alloc_set(SP_MAP);
```

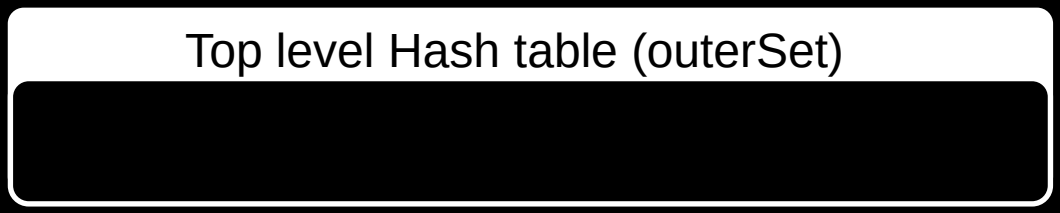
- Now we are going to go through the reads, and insert them into the edgeSet.

```
for(i = 0; i<numReads; i++){…}
```

- We have not instantiated any of the internal hash tables yet because right now because we have not gone through the reads yet
  - We will instantiate the internal hash tables only as we need them.

Building the table with set_t

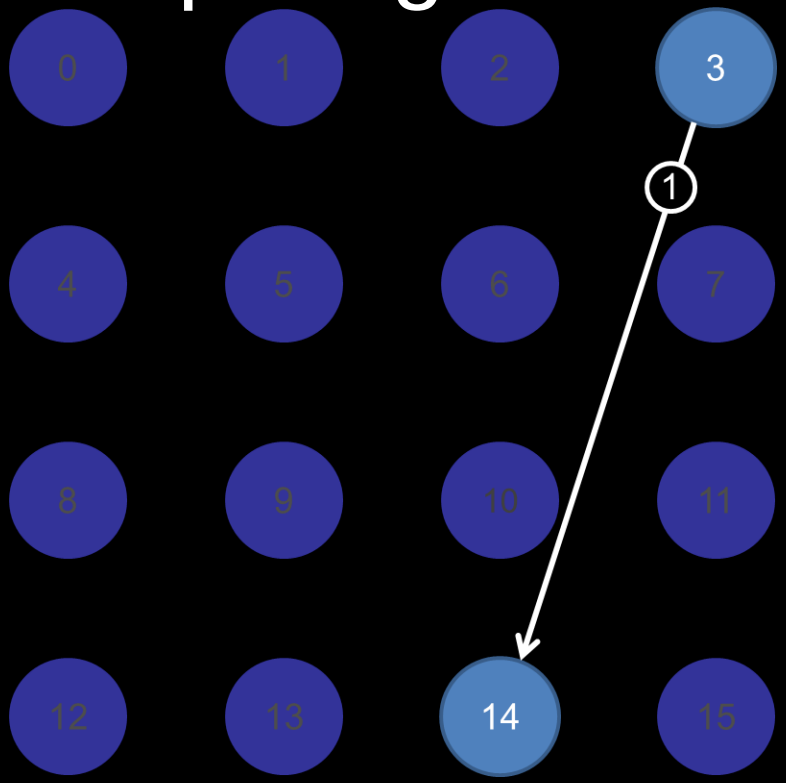





```
set_t outerSet = alloc_set(SP_MAP);
```

- First, instantiate the set. Lets call it “outer”

Preparing to insert a read into outerSet



Current Read: ATG

Read number: 0

Top level Hash table (outerSet)

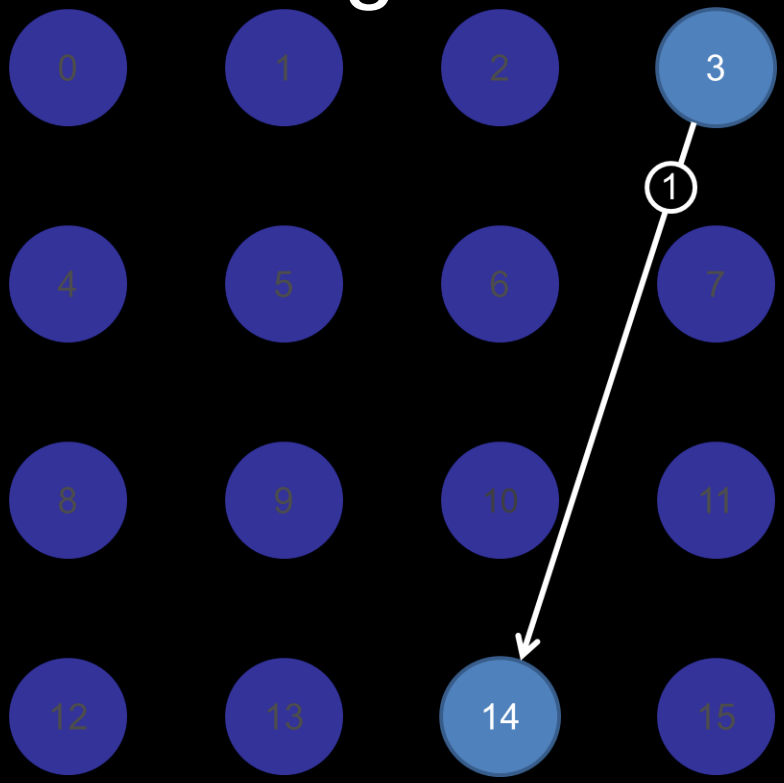
3

14

```
int pIndex = getPrefixInd("ATG");
int sIndex = getSuffixInd("ATG");
set_t innerSet;
if(contains_set(outerSet, pIndex)){
    innerSet = (set_t) mapsto_set(outerSet, pIndex);
}
else{ innerSet = alloc_set(SP_MAP); }
```

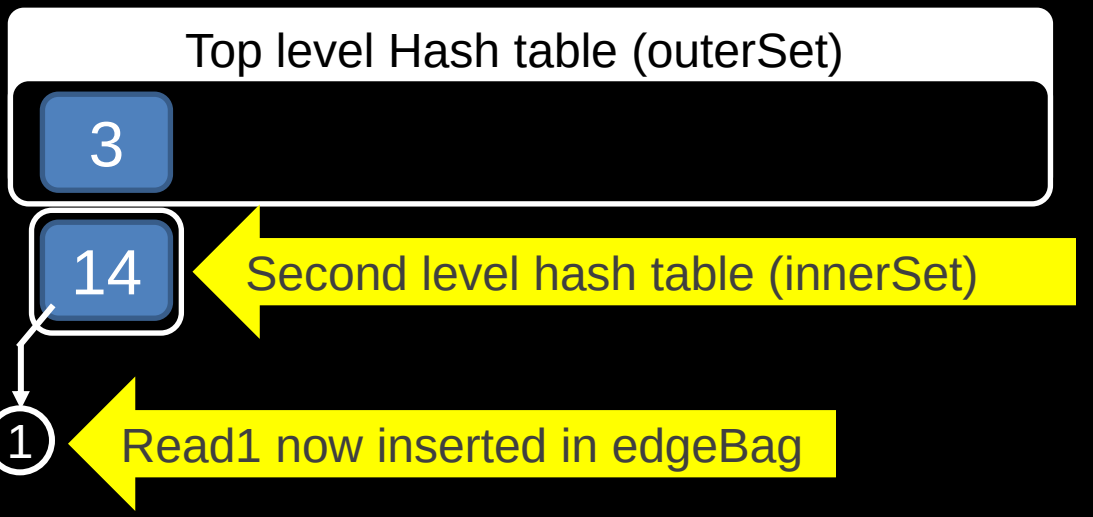
- Get the index of the prefix
- Get the index of the suffix
- Check if hash table exists for prefix
- If it does, look it up.
- If it doesn't, allocate a new one

Inserting the read into the edgeSet



Current Read: ATG

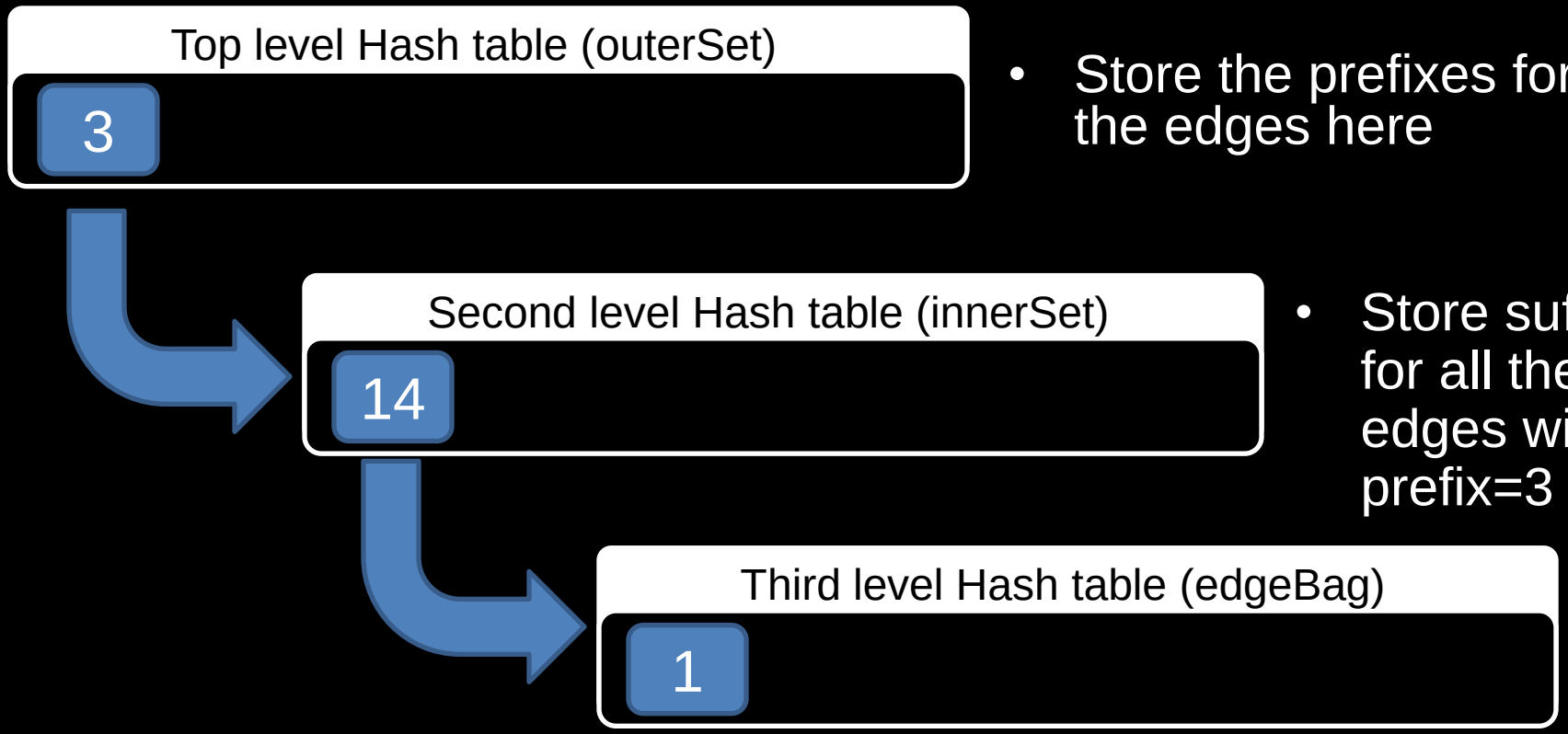
Read number: 0




```
set_t edgeBag;
if(contains_set(innerSet, sIndex)){
    edgeBag = (set_t) mapsto_set(innerSet, sIndex);
}
else{ edgeBag = alloc_set(0); }
edgeBag = put_set(edgeBag, readNumber);
innerSet = associate_set(innerSet, sIndex, edgeBag);
outerSet = associate_set(outerSet, pIndex, innerSet);
```

- edgeBag holds lists of edges for each prefix suffix pair
- Check if this suffix was used before
- If never used, create a new edgeBag
- Store the read number
- Store the edgeBag
- Store the innerSet

What did we just do?



- Store the prefixes for all the edges here
- Store suffixes for all the edges with prefix=3 here
- Multiple reads can have the same prefix and suffix. Store the index of the read (this was the first one) here

The whole shebang

```
set_t outerSet = alloc_set(SP_MAP);
for( all reads r ){
  int pIndex = getPrefixInd(r); //this is the actual index
  int sIndex = getSuffixInd(r);
  set_t innerSet;
  if( contains_set(outerSet, pIndex) ){
    innerSet = (set_t) mapsto_set(outerSet, pIndex);
  }
  else{ innerSet = alloc_set(SP_MAP); }
  set_t edgeBag;
  if( contains_set(innerSet, sIndex) ){
    edgeBag = (set_t) mapsto_set(innerSet, sIndex);
  }
  else{ edgeBag = alloc_set(0); }
  edgeBag = put_set(edgeBag, readNumber);
  innerSet = associate_set(innerSet, sIndex, edgeBag);
  outerSet = associate_set(outerSet, pIndex, innerSet);
}
```

Now that we've built an edgeset….

- Now we want to find a path
- We will start somewhere and walk from edge to edge
- We cannot reuse an edge, so we must keep track

```
set_t usedEdges = alloc_set(0);
```

- This set does not contain objects, it just contains edge indices.
- Each time we use an edge, we insert it here:

```
usedEdges = put_set(usedEdges, thisEdge);
```

Where to start?

- We need to start the path at a node with outdegree > indegree
- How do we calculate outdegree and indegree?
- Outdegree: easy. Make a function for this:
  - Each prefix corresponds to a hash table stored in outerSet. Get it out.

```
innerSet = (set_t) mapsto_set(outerSet, prefixIndex);
```

  - Each suffix corresponds to an edgeBag. Get it out.

```
edgeBag = (set_t) mapsto_set(innerSet, suffixIndex);
```

  - The outdegree is the sum of the number of members in each edgeBag in the innerSet:

```
OutDegree += size_set(edgeBag);
```

Indegree: slightly harder.

- Each indegree of a node corresponds to one read where this node was a suffix.
- Search through every innerSet in the outerSet.
- Each time an innerSet contains the suffixIndex:
  - Get out the corresponding edgeBag
  - Get the size of the edgeBag
  - Add that size to the total indegree of this node.

An easier way to get indegree/outdegree

- Record this information as you go through the reads:

```
set_t indegrees = alloc_set(SP_MAP);
set_t outdegrees = alloc_set(SP_MAP);
```

- Each time you get an edge, get its prefixIndex and suffixIndex, and store it in the sets:

```
int * counter;
if ( contains_set(indegrees, sufIndex ){
  counter = (int *) mapsto_set(indegrees, sufIndex);
}
else{  counter = new int[1]; counter[0] = 0; }
counter[0] += 1;
indegrees = associate_set(indegrees, sufIndex, counter);
```

- Each time you get an edge, get its prefixIndex, suffixIndex, store it in outdegrees or indegrees

Walking along a path of edges

- Find a node with outdegree > indegree
  - This has a specific prefix index.
- Find an edge leaving the node.

```
int suffixIndex;
int unusedEdge;
set_t innerSet = (set_t) mapsto_set( outerSet, prefixIndex);
for(i = 0; i<size_set(innerSet); i++){
        suffixIndex = innerSet[i];
        set_t edgeBag = (set_t) mapsto_set(innerSet, suffixIndex);
        for(j = 0; j<size_set(edgeBag); j++){
                int thisEdge = edgeBag[j];
                if( !contains_set(usedEdges, thisEdge){
                        unusedEdge = thisEdge;
                        break;
                }
        }
}
```

- Add that edge to the used edges,
- Add the suffixIndex to a list of nodes in your path.
  - Think carefully how you represent your paths/cycles

Completing paths and cycles

- If you come back to a node you have visited before, it is a cycle.
  - You don’t have to stop unless you are out of unused edges. Keep going – a longer path is more useful
- If there are no more unused edges from the node you are on, then it is the end of your path.
- Keep a separate list of all your paths and cycles.

Detect if your paths and cycles intersect

- You can do this if you store your paths and cycles in sets.
- Just call the contains_set() function to tell if a node you are on is in another set.

- Be careful how you store paths and cycles in a set.
  - Sets of integers are STRICTLY non-redundant. If you insert an integer that is already in the set, nothing will happen: repeating lists like cycles are bad.
  - While a set might be good for recording what nodes are used in your set, it is bad for storing repeating elements
  - If you only store nodes, then you don't know which edges you are storing. This could be a problem!

A really important point

- We are only looking at reads that overlap by exactly 15 nucleotides.
- Your test, overlap(read1, read2, 15) should return true only if the 15 nucleotides in the suffix are equal to the 15 nucleotides in the prefix.
  - You don't care if the overlap is of a different length

- There is likely to be more than one read starting at every nucleotide.
- That means that if a read R lines up at a shorter or longer overlap, another read will line up with R at exactly 15.

You have many resources around you

- Don’t work in isolation
  - Consult the other members of the class if you have questions about the project you are working on
  - Consult the other members of your group if you have questions about your report
    - Reports are intentionally cross disciplinary

- Get started now if you have not already started.
- You only have about two weeks left.

Questions