Searching Basics

- Search Algorithms
- Search Structures: Symbol Tables

Searching

- With all the vast amounts of data out there, it’s important to be able to search for specific items efficiently.
- Search algorithms are often closely tied with data structures because how data is stored affects how it can be searched.

Review of data structures…

We've discussed…

- Stacks
- Queues
- Priority Queues
- Linked Lists

None of these are built for efficient *searching* (i.e. searching for a specific element.)

What about arrays?

Linear Search on an unsorted array...

4	8	1	9	0	2	3	7	6	3

best: $O(1)$

worst: $O(N)$

Binary search on a sorted array...

0	1	2	3	3	4	6	7	8	9

best: $O(1)$

worst: $O(\log N)$

Symbol Tables

- “...an abstract mechanism where we save information (a *value*) that we can later search for and retrieve by specifying a *key*.” [1]
- Sometimes called *dictionaries* or *indices*.
- A symbol table associates *values* with *keys* (called *key-value pairs*) which get inserted into the table and can later be retrieved using the key.

Symbol Tables

- The important operations include *insert (*aka *put)* and *search* (aka *get*) but there may be other useful operations as well.
- To implement, we need…
  - an underlying data structure, and
  - definitions for the above operations (and other useful operations).

Some Applications

application	purpose of search	key	value
dictionary	find definition	word	definition
book index	find relevant pages	term	list of page numbers
file share	find song to download	name of song	computer ID
account management	process transactions	account number	transaction details
web search	find relevant web pages	keyword	list of page names
compiler	find type and value	variable name	type and value

Typical symbol-table applications

Picture from [1]

public class ST<Key, Value>		
	ST()	*create a symbol table*
void	put(Key key, Value val)	*put key-value pair into the table (remove* key *from table if value is* null*)*
Value	get(Key key)	*value paired with* key *(*null *if* key *is absent)*
void	delete(Key key)	*remove* key *(and its value) from table*
boolean	contains(Key key)	*is there a value paired with* key*?*
boolean	isEmpty()	*is the table empty?*
int	size()	*number of key-value pairs in the table*
Iterable<Key>	keys()	*all the keys in the table*

API for a generic basic symbol table

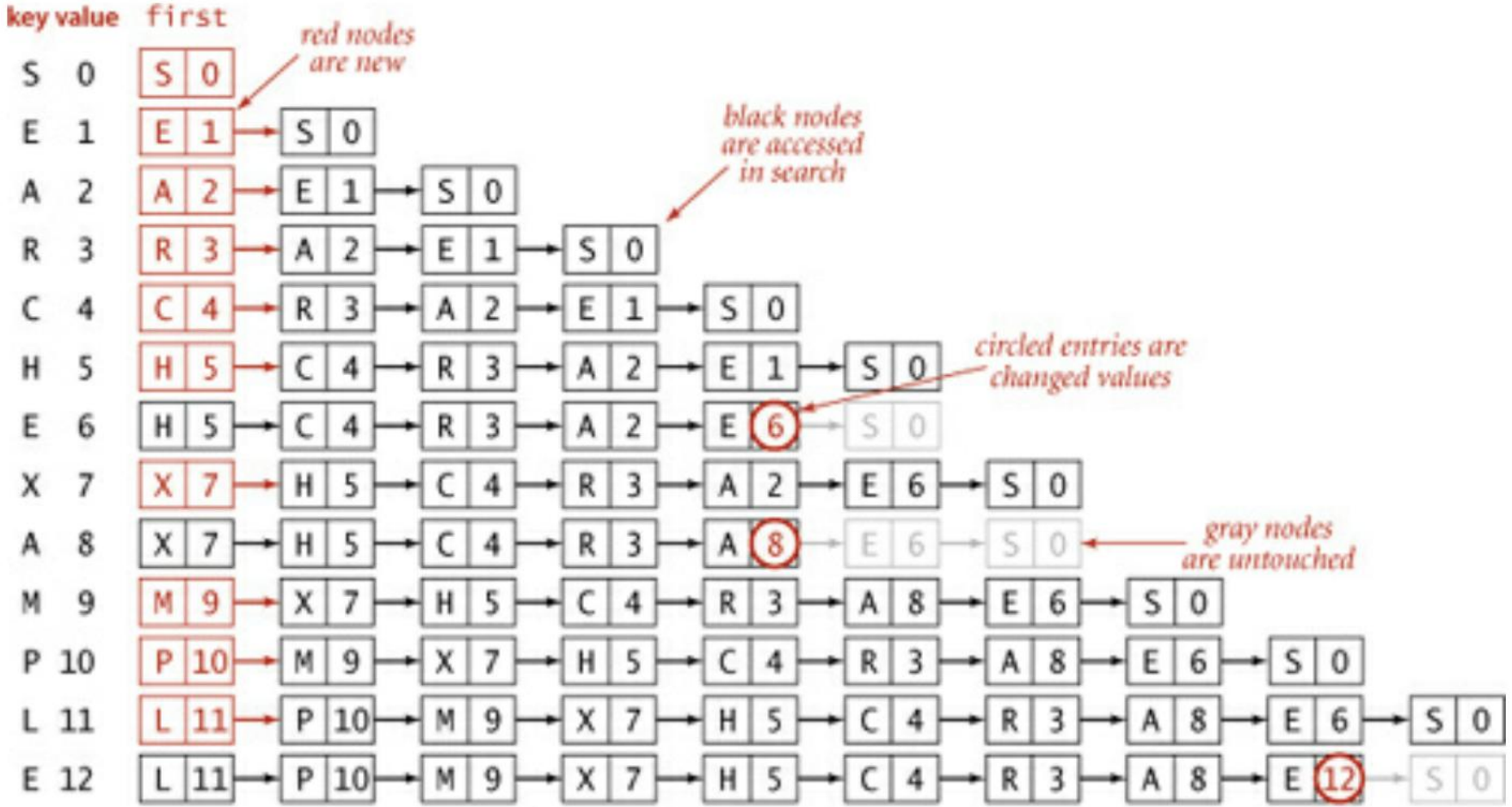
Picture from [1]

How would you implement a symbol table?

Details:

- No duplicate keys (i.e. Each key can only have one value, but that value could be a list.)
- If you try to insert (k, v), and k is already associated with w, then (k, w) is updated.

Idea 1: Use a linked list.

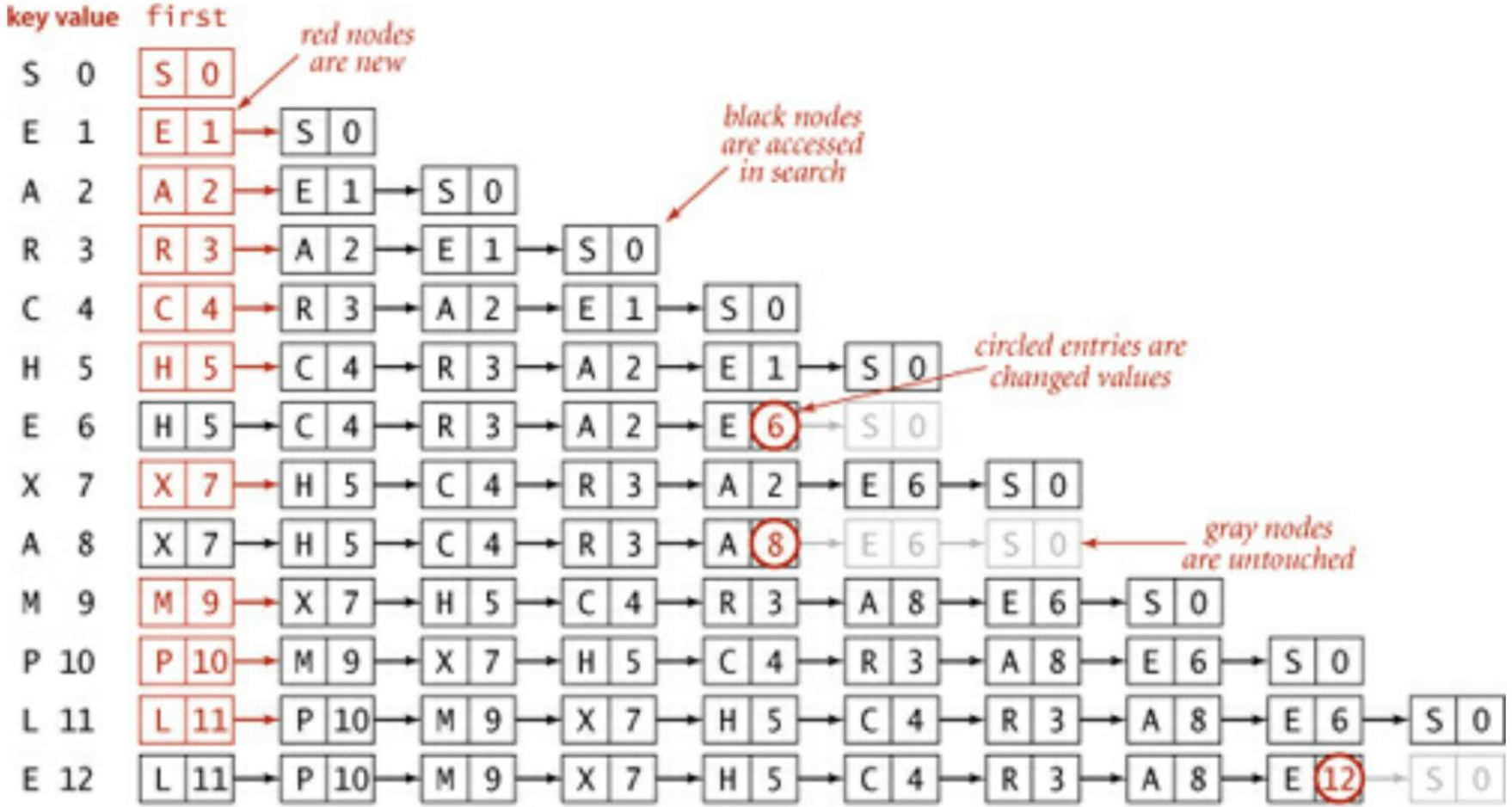


Trace of linked-list ST implementation for standard indexing client

Picture from [1]

What is the worst-case runtime for *put* and *get* if you are using a linked list and the size is N?

$O(N)$



Trace of linked-list ST implementation for standard indexing client

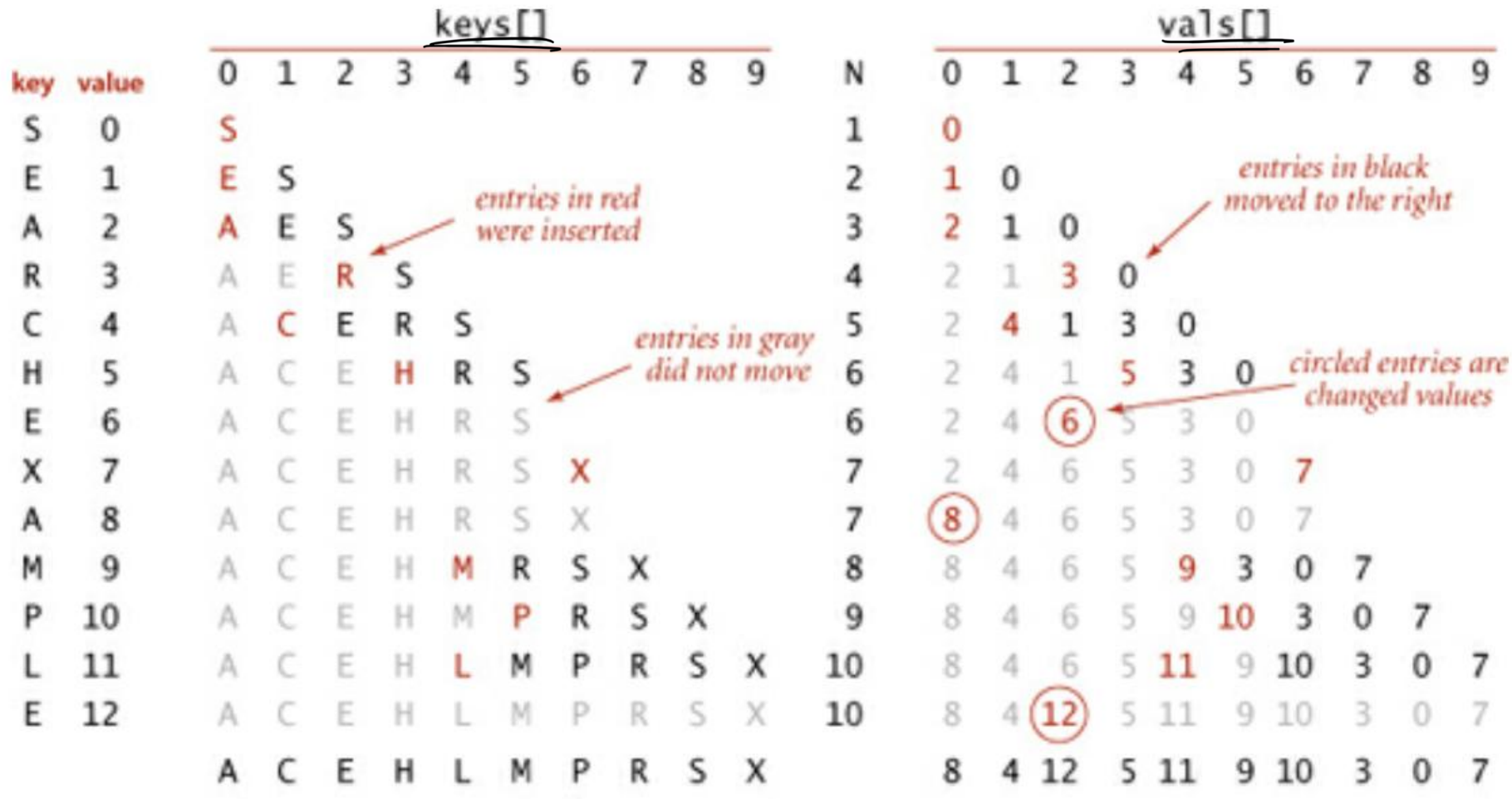
Picture from [1]

Idea 2: Use an array (or a pair of arrays).

What are some questions to consider with this approach?

Questions to consider regarding Idea 2...

- How will you manage the size when an array has a fixed capacity?
- What if you have millions of key-value pairs?
- How is the data organized?
- How are keys associated with a location in the array?



Trace of ordered-array ST implementation for standard indexing client

Picture from [1]

Ordered Symbol Table:
Keeps the Keys in order

How would this affect the runtime for the basic operations?

How would you implement this?

public class ST<Key extends Comparable<Key>, Value>		
	ST()	*create an ordered symbol table*
void	put(Key key, Value val)	*put key-value pair into the table (remove* key *from table if value is* null*)*
Value	get(Key key)	*value paired with* key *(*null *if* key *is absent)*
void	delete(Key key)	*remove* key *(and its value) from table*
boolean	contains(Key key)	*is there a value paired with* key*?*
boolean	isEmpty()	*is the table empty?*
int	size()	*number of key-value pairs*
Key	min()	*smallest key*
Key	max()	*largest key*
Key	floor(Key key)	*largest key less than or equal to* key
Key	ceiling(Key key)	*smallest key greater than or equal to* key
int	rank(Key key)	*number of keys less than* key
Key	select(int k)	*key of rank* k
void	deleteMin()	*delete smallest key*
void	deleteMax()	*delete largest key*
int	size(Key lo, Key hi)	*number of keys in* [lo..hi]
Iterable<Key>	keys(Key lo, Key hi)	*keys in* [lo..hi]*, in sorted order*
Iterable<Key>	keys()	*all keys in the table, in sorted order*

API for a generic ordered symbol table

Picture from [1]

Analysis Summary

method	order of growth of running time
put()	N
get()	$\log N$
delete()	N
contains()	$\log N$
size()	1
min()	1
max()	1
floor()	$\log N$
ceiling()	$\log N$
rank()	$\log N$
select()	1
deleteMin()	N
deleteMax()	1

BinarySearchST costs

algorithm (data structure)	worst-case cost (after N inserts)		average-case cost (after N random inserts)		efficiently support ordered operations?
	search	insert	search hit	insert	
sequential search (unordered linked list)	N	N	$N/2$	N	no
binary search (ordered array)	$\lg N$	$2N$	$\lg N$	N	yes

Cost summary for basic symbol-table implementations

Which one is BETTER?
Could we improve upon it?

Pictures from [1]

References

[1] *Algorithms, Fourth Edition;* Robert Sedgewick and Kevin Wayne (and associated slides)