

list.h

```
1 /
  *****
  **
2 File name:      list.h
3 Author:        CS, Pacific University
4 Date:         9/30/13
5 Class:        CS300
6 Assignment:    List Implementation
7 Purpose:      This file defines the constants, data structures, and
  function
8               prototypes for implementing a list data structure. In
  essence,
9               the list API is defined for other modules.
10
  *****
  */
11
12 #ifndef LIST_H_
13 #define LIST_H_
14
15 #include <stdbool.h>
16
17 #define TRUE      1
18 #define FALSE    0
19
20 // List error codes for each function to use
21
22 #define NO_ERROR          0
23
24 // list create failed
25 #define ERROR_NO_LIST_CREATE  -1
26
27 // user tried to operate on an empty list
28 #define ERROR_EMPTY_LIST     -2
29
30 // user tried to add data to a full list
31 #define ERROR_FULL_LIST      -3
32
33 // user tried to peekNext when no next existed
34 #define ERROR_NO_NEXT        -6
35
36 // user tried to peekPrev when no next existed
37 #define ERROR_NO_PREV        -7
38
39 // user tried to use current when current was not defined
40 #define ERROR_NO_CURRENT     -8
41
42 // user tried to operate on an invalid list. An invalid
43 // list may be a NULL ListPtr or contain an invalid value for numElements
```

list.h

```
44 #define ERROR_INVALID_LIST -9
45
46 // user provided a NULL pointer to the function (other than the ListPtr)
47 #define ERROR_NULL_PTR -10
48
49
50 #define NO_CURRENT -100
51 #define EMPTY_LIST 0
52 // Possible datatypes for later use
53
54 #define CHARACTER_VALUE 0
55 #define INTEGER_VALUE 1
56 #define FLOAT_VALUE 2
57
58 #define NO_CURRENT -100
59 #define EMPTY_LIST 0
60
61 // User-defined datatypes for easier reading
62
63
64 typedef short int ERRORCODE;
65
66 // The user of this data structure is only concerned with
67 // two data types: List and DATATYPE. ListElement is an internal
68 // data structure not to be directly used by the user.
69 // If the List implementation changes (to static memory, a tree, etc)
70 // ListElement will change.
71
72 typedef struct DATATYPE
73 {
74     union
75     {
76         char charValue;
77         unsigned int intValue;
78         float floatValue;
79     };
80     unsigned short whichOne;
81 } DATATYPE;
82
83 typedef struct ListElement* ListElementPtr;
84 typedef struct ListElement
85 {
86     DATATYPE data;
87     ListElementPtr psNext;
88     ListElementPtr psPrev;
89 } ListElement;
90
91
92 // A list is a dynamic data structure where the current pointer and number
```

list.h

```
93 // of elements are maintained at all times
94
95 typedef struct List* ListPtr;
96 typedef struct List
97 {
98     ListElementPtr psHead;
99     ListElementPtr psLast;
100    ListElementPtr psCurrent;
101    int numElements;
102 } List;
103
104 /
105 *
106 *           Allocation and Deallocation
107 *
108 *
109 *
110 *
111 *
112 *
113 *
114 *
115 *
116 *           Checking number of elements in list
117 *
118 *
119 *
120 *
121 *
122 *
123 *
124 *
125 *
126 *
127 *
128 *
129 *
130 *           Peek Operations
131 *
132 *
133 *
134 *
135 *
136 *
137 *
138 *
139 *
140 *
141 *
142 *
143 *
144 *
145 *
146 *
147 *
148 *
149 *
150 *
151 *
152 *
153 *
154 *
155 *
156 *
157 *
158 *
159 *
160 *
161 *
162 *
163 *
164 *
165 *
166 *
167 *
168 *
169 *
170 *
171 *
172 *
173 *
174 *
175 *
176 *
177 *
178 *
179 *
180 *
181 *
182 *
183 *
184 *
185 *
186 *
187 *
188 *
189 *
190 *
191 *
192 *
193 *
194 *
195 *
196 *
197 *
198 *
199 *
200 *
201 *
202 *
203 *
204 *
205 *
206 *
207 *
208 *
209 *
210 *
211 *
212 *
213 *
214 *
215 *
216 *
217 *
218 *
219 *
220 *
221 *
222 *
223 *
224 *
225 *
226 *
227 *
228 *
229 *
230 *
231 *
232 *
233 *
234 *
235 *
236 *
237 *
238 *
239 *
240 *
241 *
242 *
243 *
244 *
245 *
246 *
247 *
248 *
249 *
250 *
251 *
252 *
253 *
254 *
255 *
256 *
257 *
258 *
259 *
260 *
261 *
262 *
263 *
264 *
265 *
266 *
267 *
268 *
269 *
270 *
271 *
272 *
273 *
274 *
275 *
276 *
277 *
278 *
279 *
280 *
281 *
282 *
283 *
284 *
285 *
286 *
287 *
288 *
289 *
290 *
291 *
292 *
293 *
294 *
295 *
296 *
297 *
298 *
299 *
300 *
301 *
302 *
303 *
304 *
305 *
306 *
307 *
308 *
309 *
310 *
311 *
312 *
313 *
314 *
315 *
316 *
317 *
318 *
319 *
320 *
321 *
322 *
323 *
324 *
325 *
326 *
327 *
328 *
329 *
330 *
331 *
332 *
333 *
334 *
335 *
336 *
337 *
338 *
339 *
340 *
341 *
342 *
343 *
344 *
345 *
346 *
347 *
348 *
349 *
350 *
351 *
352 *
353 *
354 *
355 *
356 *
357 *
358 *
359 *
360 *
361 *
362 *
363 *
364 *
365 *
366 *
367 *
368 *
369 *
370 *
371 *
372 *
373 *
374 *
375 *
376 *
377 *
378 *
379 *
380 *
381 *
382 *
383 *
384 *
385 *
386 *
387 *
388 *
389 *
390 *
391 *
392 *
393 *
394 *
395 *
396 *
397 *
398 *
399 *
400 *
401 *
402 *
403 *
404 *
405 *
406 *
407 *
408 *
409 *
410 *
411 *
412 *
413 *
414 *
415 *
416 *
417 *
418 *
419 *
420 *
421 *
422 *
423 *
424 *
425 *
426 *
427 *
428 *
429 *
430 *
431 *
432 *
433 *
434 *
435 *
436 *
437 *
438 *
439 *
440 *
441 *
442 *
443 *
444 *
445 *
446 *
447 *
448 *
449 *
450 *
451 *
452 *
453 *
454 *
455 *
456 *
457 *
458 *
459 *
460 *
461 *
462 *
463 *
464 *
465 *
466 *
467 *
468 *
469 *
470 *
471 *
472 *
473 *
474 *
475 *
476 *
477 *
478 *
479 *
480 *
481 *
482 *
483 *
484 *
485 *
486 *
487 *
488 *
489 *
490 *
491 *
492 *
493 *
494 *
495 *
496 *
497 *
498 *
499 *
500 *
501 *
502 *
503 *
504 *
505 *
506 *
507 *
508 *
509 *
510 *
511 *
512 *
513 *
514 *
515 *
516 *
517 *
518 *
519 *
520 *
521 *
522 *
523 *
524 *
525 *
526 *
527 *
528 *
529 *
530 *
531 *
532 *
533 *
534 *
535 *
536 *
537 *
538 *
539 *
540 *
541 *
542 *
543 *
544 *
545 *
546 *
547 *
548 *
549 *
550 *
551 *
552 *
553 *
554 *
555 *
556 *
557 *
558 *
559 *
560 *
561 *
562 *
563 *
564 *
565 *
566 *
567 *
568 *
569 *
570 *
571 *
572 *
573 *
574 *
575 *
576 *
577 *
578 *
579 *
580 *
581 *
582 *
583 *
584 *
585 *
586 *
587 *
588 *
589 *
590 *
591 *
592 *
593 *
594 *
595 *
596 *
597 *
598 *
599 *
600 *
601 *
602 *
603 *
604 *
605 *
606 *
607 *
608 *
609 *
610 *
611 *
612 *
613 *
614 *
615 *
616 *
617 *
618 *
619 *
620 *
621 *
622 *
623 *
624 *
625 *
626 *
627 *
628 *
629 *
630 *
631 *
632 *
633 *
634 *
635 *
636 *
637 *
638 *
639 *
640 *
641 *
642 *
643 *
644 *
645 *
646 *
647 *
648 *
649 *
650 *
651 *
652 *
653 *
654 *
655 *
656 *
657 *
658 *
659 *
660 *
661 *
662 *
663 *
664 *
665 *
666 *
667 *
668 *
669 *
670 *
671 *
672 *
673 *
674 *
675 *
676 *
677 *
678 *
679 *
680 *
681 *
682 *
683 *
684 *
685 *
686 *
687 *
688 *
689 *
690 *
691 *
692 *
693 *
694 *
695 *
696 *
697 *
698 *
699 *
700 *
701 *
702 *
703 *
704 *
705 *
706 *
707 *
708 *
709 *
710 *
711 *
712 *
713 *
714 *
715 *
716 *
717 *
718 *
719 *
720 *
721 *
722 *
723 *
724 *
725 *
726 *
727 *
728 *
729 *
730 *
731 *
732 *
733 *
734 *
735 *
736 *
737 *
738 *
739 *
740 *
741 *
742 *
743 *
744 *
745 *
746 *
747 *
748 *
749 *
750 *
751 *
752 *
753 *
754 *
755 *
756 *
757 *
758 *
759 *
760 *
761 *
762 *
763 *
764 *
765 *
766 *
767 *
768 *
769 *
770 *
771 *
772 *
773 *
774 *
775 *
776 *
777 *
778 *
779 *
780 *
781 *
782 *
783 *
784 *
785 *
786 *
787 *
788 *
789 *
790 *
791 *
792 *
793 *
794 *
795 *
796 *
797 *
798 *
799 *
800 *
801 *
802 *
803 *
804 *
805 *
806 *
807 *
808 *
809 *
810 *
811 *
812 *
813 *
814 *
815 *
816 *
817 *
818 *
819 *
820 *
821 *
822 *
823 *
824 *
825 *
826 *
827 *
828 *
829 *
830 *
831 *
832 *
833 *
834 *
835 *
836 *
837 *
838 *
839 *
840 *
841 *
842 *
843 *
844 *
845 *
846 *
847 *
848 *
849 *
850 *
851 *
852 *
853 *
854 *
855 *
856 *
857 *
858 *
859 *
860 *
861 *
862 *
863 *
864 *
865 *
866 *
867 *
868 *
869 *
870 *
871 *
872 *
873 *
874 *
875 *
876 *
877 *
878 *
879 *
880 *
881 *
882 *
883 *
884 *
885 *
886 *
887 *
888 *
889 *
890 *
891 *
892 *
893 *
894 *
895 *
896 *
897 *
898 *
899 *
900 *
901 *
902 *
903 *
904 *
905 *
906 *
907 *
908 *
909 *
910 *
911 *
912 *
913 *
914 *
915 *
916 *
917 *
918 *
919 *
920 *
921 *
922 *
923 *
924 *
925 *
926 *
927 *
928 *
929 *
930 *
931 *
932 *
933 *
934 *
935 *
936 *
937 *
938 *
939 *
940 *
941 *
942 *
943 *
944 *
945 *
946 *
947 *
948 *
949 *
950 *
951 *
952 *
953 *
954 *
955 *
956 *
957 *
958 *
959 *
960 *
961 *
962 *
963 *
964 *
965 *
966 *
967 *
968 *
969 *
970 *
971 *
972 *
973 *
974 *
975 *
976 *
977 *
978 *
979 *
980 *
981 *
982 *
983 *
984 *
985 *
986 *
987 *
988 *
989 *
990 *
991 *
992 *
993 *
994 *
995 *
996 *
997 *
998 *
999 *
1000 *
```

list.h

```
133 // requires: List is not empty
134 // results: The value of the current element is
135 // returned through the argument list
136 // IMPORTANT: Do not change current
137
138 ERRORCODE lstPeekPrev (ListPtr, DATATYPE *);
139 // requires: List contains two or more elements and
140 // current is not the first element
141 // results: The data value of current's predecessor is returned
142 // through the argument list.
143 // IMPORTANT: Do not change current
144
145 ERRORCODE lstPeekNext (ListPtr, DATATYPE *);
146 // requires: List contains two or more elements and
147 // current is not the last element
148 // results: The data value of current's successor is returned
149 // through the argument list.
150 // IMPORTANT: Do not change current
151
152 /
    *****
    *
153 *           Retrieving values and updating current
154 *****
    */
155
156 ERRORCODE lstFirst(ListPtr, DATATYPE *);
157 // requires: List is not empty
158 // results: The value of the first element is returned
159 // IMPORTANT: Current is changed to first if it exists
160
161 ERRORCODE lstLast(ListPtr, DATATYPE *);
162 // requires: List is not empty
163 // results: The value of the last element is returned
164 // IMPORTANT: Current is changed to last if it exists
165
166 ERRORCODE lstNext(ListPtr, DATATYPE *);
167 // requires: List is not empty, and current is not past the end of the list
168 // results: The value of the current element is returned
169 // IMPORTANT: Current is changed to the successor of the current element
170
171 ERRORCODE lstPrev(ListPtr, DATATYPE *);
172 // requires: List is not empty, and current is not past the first of the
    list
173 // results: The value of the current element is returned
174 // IMPORTANT: Current is changed to previous if it exists
175
176 /
    *****
```

list.h

```
*
177 *           Insertion, Deletion, and Updating
178 *****
*/
179
180 ERRORCODE lstDeleteCurrent (ListPtr, DATATYPE *);
181 // requires: List is not empty
182 // results: The current element is deleted and its
183 // successor and predecessor become each
184 // others successor and predecessor. If the
185 // deleted element had a predecessor, then
186 // make it the new current element; otherwise,
187 // make the first element current if it exists.
188 // The deleted element is returned through the argument list.
189
190 ERRORCODE lstInsertAfter (ListPtr, DATATYPE);
191 // requires: List is not full
192 // results: if the list is not empty, insert the new
193 // element as the successor of the current
194 // element and make the inserted element the
195 // current element; otherwise, insert element
196 // and make it current.
197
198 ERRORCODE lstInsertBefore (ListPtr, DATATYPE);
199 // requires: List is not full
200 // results: If the list is not empty, insert the new
201 // element as the predecessor of the current
202 // element and make the inserted element the
203 // current element; otherwise, insert element
204 // and make it current.
205
206 ERRORCODE lstUpdateCurrent (ListPtr, DATATYPE);
207 // requires: List is not empty
208 // results: The value of ListElement is copied into the current element
209
210 /
    *****
    *
211 *           List Testing
212 *****
*/
213
214 ERRORCODE lstHasNext (ListPtr, bool *);
215 // results: Returns true if there are more elements when traversing
216 // the list in a forward direction; otherwise, false is returned.
217
218 ERRORCODE lstHasPrev(ListPtr, bool *);
219 // results: Returns true if the current node has a
220 // predecessor; otherwise, false is returned
```

list.h

```
221  
222 #endif /* LIST_H_ */  
223  
224
```